Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Sistema de detección y clasificación de obstáculos para JetBot basado en redes neuronales

Autor: Alejandro Mendoza Barrionuevo

Tutor: Daniel Gutierrez Reina

Dpto. Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Sistema de detección y clasificación de obstáculos para JetBot basado en redes neuronales

Autor:
Alejandro Mendoza Barrionuevo

Tutor:
Daniel Gutierrez Reina
Profesor Contratado

Dpto. Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado: Sistema de detección y clasificación de obstáculos para JetBot basado en redes neuronales

Autor: Alejandro Mendoza Barrionuevo
Tutor: Daniel Gutierrez Reina

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Han sido muchas las personas que me han ayudado a llegar a este punto, entregar un Trabajo de Fin de Grado que pone el cierre a mi periodo como estudiante del grado en Ingeniería Electrónica, Robótica y Mecatrónica.

Durante los largos años de aprendizaje he tenido el apoyo y comprensión incondicional de toda mi familia, quienes me han ayudado con su cariño, consejo y dedicación especialmente mi madre y mi hermana, las cuales se han quedado en los momentos más difíciles y me han dado las condiciones necesarias para estudiar. Por quienes han estado, y por quienes ya no están, gracias de corazón.

Todos los compañeros y amigos que he podido conocer durante el estudio de esta carrera y con los que he compartido tantos momentos, y especialmente a Eduardo Moscosio, mi fiel compañero de batallas, quien me ha dado ánimos y ayuda tanto dentro como fuera del ámbito universitario, *till the end of the line, pal*.

Agradecer también a todos los trabajadores de la universidad, desde aquellos profesores que de verdad sí sienten su profesión hasta el personal de la propia Escuela de Ingeniería, que con su labor permiten que tantas personas puedan seguir aprendiendo y que todo esto sea posible.

Por último, dar las gracias a mi tutor, Daniel Gutierrez, por haber confiado en mí para la realización de este proyecto, quien me ha guiado y orientado a lo largo del proceso, así como sus compañeros de laboratorio.

Muchas gracias a todos, este logro es tan mío como vuestro.

Alejandro Mendoza Barrionuevo
Sevilla, 2021

Resumen

Este proyecto pretende abordar la creación de un modelo de red neuronal capaz de detectar y clasificar objetos. Para ello se hará uso de un robot terrestre con cámara y un mini-computador a bordo, en el cual será ejecutada la red. El robot elegido será el Kit de IA de JetBot de Waveshare, el cual cuenta con espacio para la plataforma Nvidia Jetson Nano. En la Jetson será instalado el paquete de sistema operativo JetPack, corriendo sobre la base de Ubuntu 18.04, el cual será preparado y configurado con todas las herramientas para su uso en visión por computador.

En lugar de crear la red desde cero, se realizará un proceso de *transfer learning* con YOLO, la cual se caracteriza por su velocidad de ejecución y su precisión sin dejar de lado la rapidez, permitiendo facilitar el proceso y reducir el tiempo de entrenamiento. Los objetos serán definidos como cajas de distintos tamaños y colores hechos específicamente para este proyecto, y se realizará un banco de imágenes propio con el que poder proceder al entrenamiento de YOLO.

Tras la finalización de este proceso, se adoptará una postura crítica para analizar los resultados obtenidos, entre los que se incluirán gráficas, cálculos de parámetros que indiquen el acierto, precisión y fiabilidad de la red, así como imágenes y vídeos que demuestren su ejecución de forma más visual.

El trabajo realizado será usado como analogía para poder desarrollar en un futuro un vehículo de navegación acuático que sea capaz de detectar obstáculos y realizar un manejo autónomo.

Abstract

This project aims to create a neural network model capable of detecting and classifying objects. This will be done using a terrestrial robot with a camera and a mini-computer on board, in which the network will be executed. The chosen robot will be the JetBot AI Kit from Waveshare, which has space for the Nvidia Jetson Nano platform. The JetPack operating system package will be installed on the Jetson, running on Ubuntu 18.04, which will be prepared and configured with all the tools for its use in computer vision.

Instead of creating the network from scratch, a transfer learning process will be performed with YOLO. This is characterized by its speed of execution and accuracy without neglecting the speed, allowing to facilitate the process and to reduce training time. Boxes will be used as objects. These are different sizes and colors made specifically for this Project and an image bank will be created in order to proceed with the training of YOLO.

After the end of this process, a critical stance will be adopted to analyze the results obtained. These will include graphs and parameter calculations that indicate the accuracy, precision and reliability of the network, as well as images and videos that demonstrate its execution in a more visual way.

The work carried out will be used as an analogy for the future development of an aquatic navigation vehicle capable of detecting obstacles and performing autonomous operations.

Índice Abreviado

Índice

1 Introducción

"The important thing is not to stop questioning. Curiosity has its own reason for existence."

—Albert Einstein

La inteligencia artificial se ha convertido, probablemente, en una de las mayores revoluciones recientes. Tal es su capacidad y su complejidad que importantes autores la catalogan como *«el desafío del siglo»*, ya no únicamente por sus grandes beneficios, sino también por los nuevos debates que se abren y esconden tras ella. Desde el enorme manejo, almacenamiento y correlación de información que son capaces de alcanzar, en contraposición a los seres humanos, hasta la toma de decisiones trascendentales y sus consecuencias en la intimidad de los individuos, son aspectos a destacar por los sectores más reacios a esta nueva era.

Para poder crear una opinión más independiente es necesario indagar en qué tipos de inteligencia artificial existen, qué datos manejan, y qué son capaces de hacer, lo cual se pondrá en contexto en el segundo capítulo. Pero antes es necesario centrarse en varios aspectos de este proyecto en concreto.

1.1 Motivaciones y objetivos

A lo largo del grado universitario se han adquirido conocimientos sobre electrónica y control, pero no se han tratado temas más actuales como pueden ser la automatización de tareas, visión por computador o inteligencia artificial, por lo que era un momento ideal para indagar en este tema y adquirir competencias muy interesantes de cara al futuro. Personalmente, este proyecto se enfrenta como un reto personal, el cual se espera y desea llevar a cabo de la forma más profesional y competente posible.

El proyecto a realizar estaría enfocado en vehículos a motor flotantes, los cuales dispondrían de cámaras capaces de reconocer el entorno, y un controlador para variar la velocidad y dirección del propio vehículo. Para ello sería necesario un banco de pruebas para la monitorización de recursos hídricos. El objetivo sería la creación e implementación de un algoritmo de inteligencia artificial basada en red neuronal que maneje la información aportada por dichas cámaras, reconociendo obstáculos alrededor del vehículo, como pueden ser boyas, piedras, salientes, o incluso otros barcos.

Cuando la IA detectara el obstáculo, la información debe ser enviada al controlador, el cual se encargaría de gestionar la incidencia y proporcionar las señales de control necesarias para conseguir la esquiva y superar el inconveniente de forma satisfactoria.

La red neuronal, además de detectar el obstáculo, debe ser capaz de clasificarlos, a partir de una base de datos de los distintos obstáculos posibles. Tras esto podrían ser ordenados y priorizados, por ejemplo, según su grado de peligrosidad, su tamaño, su naturaleza, etc.

Como no se dispone de las infraestructuras, permisos, ni material requerido para la creación de un proyecto de estas características, las pruebas se realizarán mediante el Kit de IA de JetBot de Waveshare, el cual tendrá integrada una NVIDIA Jetson Nano como computadora, donde estará implementada la red neuronal que detectará y clasificará los obstáculos observados a través de un módulo de cámara diseñado para Raspberry Pi. Dichos obstáculos serán simulados mediante bloques, realizados con cajas que serán envueltas en papeles de distintos colores.

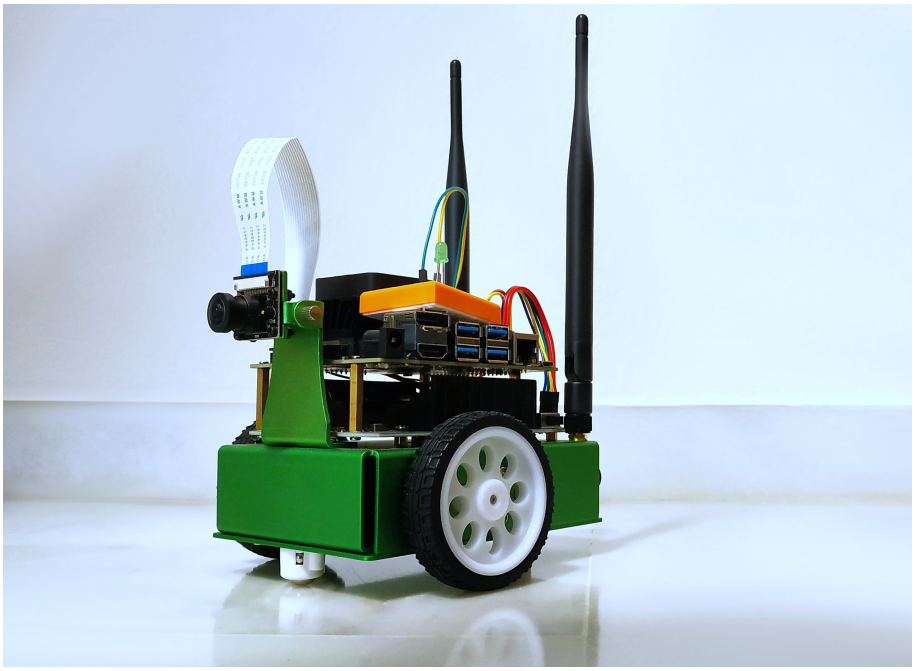


Figura 1.1 Vehículo móvil JetBot usado para el proyecto, con cámara y Jetson Nano incorporados.

Recapitulando todo lo citado anteriormente, se reunirán los objetivos listados a modo de resumen:

- Iniciación en el campo de conocimiento de Inteligencia Artificial, *Machine Learning* y redes neuronales.
- Estudio de los paquetes y librerías necesarias para llevar a cabo el proyecto
- Preparación de la Jetson Nano para su uso en visión por computador y *deep learning* mediante la instalación del sistema y paquetes necesarios
- Creación de los obstáculos adaptados al proyecto, a fin de, posteriormente, realizar un *dataset* adecuado para el reentrenamiento de la red neuronal YOLO
- Llevar a cabo el proceso de reentrenamiento hasta obtener unos pesos válidos y funcionales para el cometido del proyecto
- Análisis de los resultados y ejecución de diversas pruebas con el objeto de determinar la bondad de la red

Todos estos puntos serán descritos en el siguiente apartado de forma más detallada, para luego ser profundizados y afianzados a lo largo de los futuros capítulos .

1.2 Procedimientos

Los pasos a seguir para la realización de este proyecto deben ser estructurados en orden ascendente de dificultad. En primer lugar será necesaria un aprendizaje sobre los conceptos básicos de algoritmos de inteligencia artificial, redes neuronales, *machine learning*, detección de obstáculos, etc.

Una vez adquiridos estos conocimientos, se debe proceder a investigar acerca de los programas y herramientas mayoritariamente usadas en este ámbito, como pueden ser el lenguaje de programación Python y los paquetes TensorFlow, Keras, OpenCV, Numpy, etc. Tras la intensa exploración del entorno de investigación, será necesario aplicarlo a la práctica, y proceder a su instalación en la herramienta de trabajo a usar, la Jetson Nano.

En esta computadora se instalará el sistema JetPack, junto con todas las herramientas mencionadas anteriormente para el correcto desarrollo del estudio, y, tras ello, se procederá a la creación de un *dataset* de los objetos, formados por cajas, mediante la programación de los códigos necesarios para realizar la tarea, y con un resultado válido para poder entrenar la red YOLO.

Tras su elaboración, será necesario el proceso de etiquetado de las imágenes y de realización de las *bounding boxes* en cada una de las fotografías destinadas a entrenamiento. La red neuronal asignada debe ser entrenada de forma correcta hasta lograr unos porcentajes de aciertos y precisión dentro de los márgenes válidos para la tesitura.

A continuación, tras una intensa comprobación de que todo se ejecuta de forma correcta y de un análisis de los resultados, se procederá a su implementación en el vehículo terrestre para la simulación de situaciones más realistas, debiendo ser capaz de detectar y clasificar los obstáculos existentes en el entorno únicamente a partir de visión artificial.

Aunque sería interesante la fusión con el control del vehículo móvil, el proceso de evitación de los obstáculos queda fuera de las competencias de este proyecto, por lo que se aspira a que alguien lo continúe y realice en un futuro. Para finalizar, los resultados serán expuestos mediante tablas comparativas, gráficas y vídeos que demuestren cómo de correcto es su funcionamiento y hasta qué punto puede ser aplicado en un sistema real.

1.3 Inteligencia Artificial

Definir con exactitud la idea de IA es algo complicado, ya que se trata de una cuestión compleja. Simplificando en gran medida se podría decir que es la capacidad de los ordenadores para realizar de forma automática tareas que normalmente llevan a cabo personas gracias a su comprensión, razonamiento y conocimientos. Mediante el uso de algoritmos, la máquina es capaz de aprender de los datos y utilizar lo aprendido para tomar decisiones tal y como lo haría un ser humano [55].

Una de las principales ventajas de las inteligencia artificial respecto a la inteligencia humana es que no necesitan descansar, y las cantidades de información que pueden considerar y manejar de forma simultánea es abrumadora en comparación con un humano. Es por ello que cada día son más solicitadas e importantes, y son, sin ninguna duda, una de las claves del éxito en el futuro. Además, en un mundo donde el error tiene consecuencias tan caras es de vital importancia minimizarlo, un aspecto donde las máquinas también ganan la partida a los seres humanos.

Como se ha señalado en el apartado previo, la tecnología de inteligencia artificial no es únicamente una idea de futuro, sino también de presente. Actualmente los humanos ya disfrutamos de algunas de las muchas virtudes de inteligencias artificiales, y de la simplicidad y mejora en la eficiencia que estas pueden aportar a nuestras vidas; sin embargo, debido a su gran crecimiento es necesario

también vigilar los posibles inconvenientes que puedes venir implícitos con esta progresión.

Hay quienes directamente se preguntan si la inteligencia artificial es incompatible con la privacidad [44], y es que la gigante cantidad de datos privados que estas manejan puede ser abrumadora, y recuerda a futuros distópicos altamente atados a la tecnología, vistos en numerosas ocasiones tanto en el cine como en series de televisión. Todo el mundo ha experimentado ciertas situaciones en las que duda de si su privacidad está siendo comprometida: ya sea navegando por internet, donde buscadores como Google predicen nuestras búsquedas de forma casi telepática; o comprando en páginas como Amazon, donde se sugieren con gran acierto productos relacionados con nuestros intereses e inclinaciones.

Las empresas no pueden leernos el pensamiento, al menos de momento, pero sí pueden adquirir grandes conocimientos acerca de nosotros a través de nuestras interacciones y *likes* en redes sociales, nuestras búsquedas y *clicks* o las famosas *cookies* de navegación, en un amplio espacio como es internet en el que hasta el más mínimo movimiento está rastreado. Con todas estas acciones se van dejando rastros, como si de pequeñas migas de pan se trataran, y usando técnicas de *Big Data* se consigue darle un sentido a toda esta información, logrando una simbiosis entre los posibles intereses de las empresas y de los usuarios. Una vez llegados a este punto, los clientes pueden ver estas técnicas como un avance, o verlas como un atropello a su privacidad, preguntándose si realmente ver, por ejemplo, anuncios que les interesan es realmente beneficio para el consumidor o únicamente para favorecer el consumo y el enriquecimiento de la propia corporación.

Es tan grande la importancia y el valor de los datos recopilados que muchos lo catalogan como "el nuevo petróleo" [11]. La transformación digital lleva a las compañías a superar fronteras de conocimiento acerca de los consumidores que antes era incapaz de captar. *Big Data*, *Internet of Things*, *Big Data*, *Machine Learning*, *Deep Learning*, y un sinfín de nuevas tecnologías que se usan para gestionar la información recopilada, y es que, en estos tiempos, tener información y no administrarla de forma inteligente supone una gran desventaja frente a la competencia. Desde recoger las características de personas que entran a un centro comercial como pueden ser su edad o género, hasta algoritmos capaces de predecir en qué lugar y cuándo es más probable que se produzcan accidentes de tráfico.

Pero la recolección de información no siempre se realiza de forma legal, por ejemplo, mediante el espionaje de conversaciones. Muestra de ello es la multa que interpuso la Agencia Española de Protección de Datos (AEPD) a LaLiga [48], quien, a través de su aplicación para móviles, hacía uso del micrófono y de la geolocalización para revelar la emisión de televisión de bares y locales que estuvieran emitiendo los partidos de fútbol de forma ilegítima. El recurso que LaLiga presentó afirmaba que los usuarios accedían a la autorización de los permisos para acceder a la funcionalidad del micrófono, y además, manifestó que la tecnología usada fue diseñada para crear una huella acústica concreta (*fingerprint*). Esta huella únicamente incluía un 0.75 % del sonido captado por el micrófono, despreciando el resto de la información, y que era sometida en el mismo momento a una serie de complejas transformaciones, por lo que no se podía conocer el contenido de las conversaciones ni reconocer las identidades de los hablantes.

Es por ello que no son pocos los usuarios que desconfían de los dispositivos con micrófonos, o de aquellos que están permanentemente escuchando, como puede ser el asistente de Google. La compañía asegura que únicamente se activa cuando se menciona su ya conocido *Ok Google*, pero para que esto ocurra debe estar vigilante todo el rato, lo que genera desconfianza. Es más, en julio de 2019, debido a una serie de grabaciones filtradas por una televisión belga, Google fue obligada a admitir que el 0.2 % de las conversaciones con su asistente virtual eran escuchadas por "expertos del lenguaje" contratados por la compañía para mejorar la calidad del servicio [34], lo que implicaba que los diálogos no eran completamente privados. La empresa de Mountain View se defendió alegando que exclusivamente se transcribían conversaciones dirigidas directamente a su

asistente, sin embargo, el canal belga VRT NWS fue capaz de determinar direcciones postales e información privada a partir de las grabaciones. Además, los trabajadores destinados a escuchar las conversaciones eran trabajadores externos, es decir, ni siquiera pertenecían a la propia firma. Situaciones parecidas vivieron Apple con su asistente Siri y Amazon con Alexa, los cuales también tuvieron que pedir disculpas públicas por escuchas [36]. Y es que la tecnología de reconocimiento de voz avanza a un ritmo frenético, llegando a unos niveles de precisión del 95 % de comprensión del lenguajes en casos como Google, prácticamente como un ser humano [50].

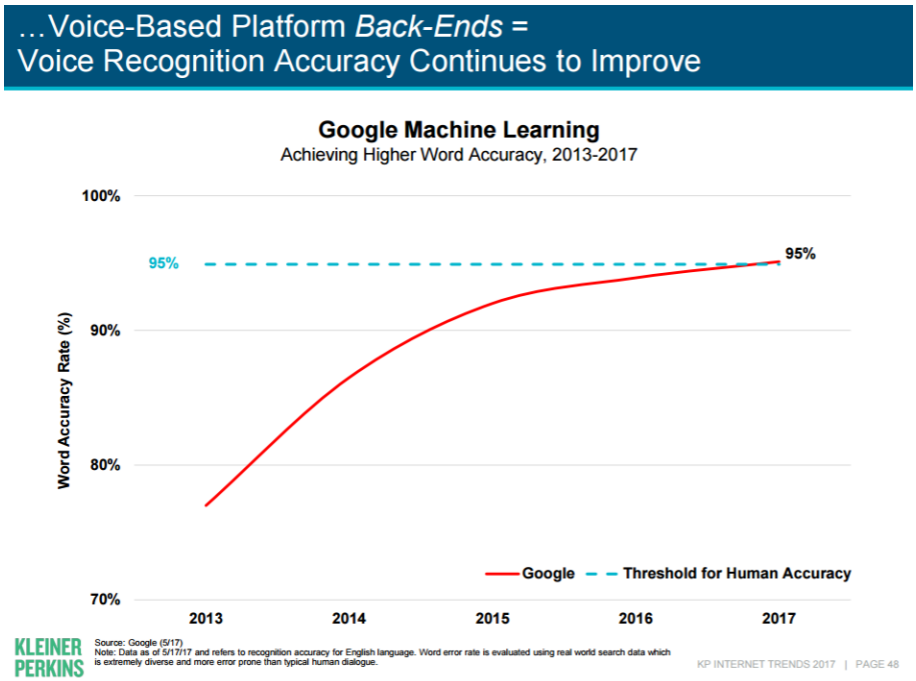


Figura 1.2 Porcentaje de precisión en el reconocimiento de voz de Google[1].

En esta coyuntura, resulta necesario preguntarse por qué son tan importantes los datos, si debe ser prioritaria su protección, o qué utilidad pueden tener más allá de la publicidad personalizada. Las respuestas pueden ser diversas, pero se puede llegar a utilizar esta información masiva incluso para influir políticamente, mostrando *fake news* según el sesgo cognitivo del usuario, pudiendo llegar a alterar la inclinación de votos, lo cual podría convertirse en un arma política determinante. Es por ello que estudios recientes afirman que las consecuencias pueden ser graves principalmente por dos motivos: en primer lugar por la escasez de conocimiento que se tiene acerca de este tema y la falta de regulación, y segundo porque este desconocimiento puede dar lugar a la desconfianza y no constatar los beneficios que la inteligencia artificial puede llegar a ofrecer [7]. La principal recomendación de los expertos es la colaboración entre legisladores y los programadores, tomando siempre en consideración tanto los puntos positivos como los negativos de las inteligencias artificiales.

Con la información contenida en este apartado se ha querido formar una opinión independiente sobre esta reciente revolución que cada día adquiere más peso en la industria. Tras mostrar en primera línea los peligros que puede conllevar el mal uso de la inteligencia artificial, el siguiente paso lógico sería presentar los maravillosos y positivos beneficios que puede aportar la inteligencia de las máquinas a la sociedad, asunto que se abordará en el Capítulo 2 sobre el Estado del Arte.

[1] Fuente: *https://miro.medium.com/max/1894/1*7epVADGxoE_R0G1Kb2_t8A.png*

2 Estado del Arte

"It's not man versus machine; it's man with machine versus man without. Data and intuition are like horse and rider, and you don't try to outrun a horse; you ride it."

—Pedro Domingos

En este capítulo se hará un repaso histórico de la evolución de las redes neuronales y su importancia en la actualidad, así como ejemplos realistas donde se ha aplicado esta tecnología. Se hará una introducción de las diferentes redes neuronales de detección que han ido evolucionando a lo largo de la historia reciente, así como las existentes en la actualidad y se pondrá en contexto los proyectos más disruptivos y ambiciosos hasta la fecha, incluyendo algunos similares al tratado en este trabajo. Con esto se pretende comprender más en profundidad los conceptos necesarios a aplicar y obtener un conocimiento general en este ámbito.

2.1 Historia de la detección de objetos y redes neuronales populares.

La detección de objetos y obstáculos por visión artificial es una ciencia que está viviendo actualmente las mayores evoluciones de su historia, teniendo una atención sin precedentes de gran parte de los grupos de investigación. Se trata de una difícil tarea que consiste en detectar diferentes clases para las que ha sido entrenada (personas, animales, coches, errores...) dentro de imágenes digitales o vídeos. Su objetivo se puede resumir en una sencilla pregunta: *«¿Qué objetos están y dónde?»* [68].

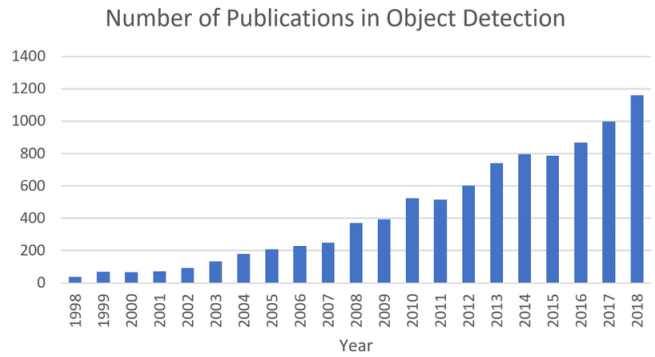


Figura 2.1 Incremento en el número de publicaciones en detección de objetos desde 1998 a 2018, según búsquedas en Google *Scholar* [1].

Implica una labor que, a simple vista, puede parecer sencilla, ya que los humanos tenemos la capacidad de detectar objetos con gran facilidad, saber si son grandes o pequeños, conocer las características de una imagen, si esta está oscura o borrosa, o incluso deducir situaciones, por ejemplo saber que alguien permanece escondido detrás de una columna aunque sólo se le vean los pies o la cabeza [10].

Esto que para nosotros resulta algo sencillo, para los algoritmos de *Machine Learning* es una ardua tarea. No pueden reconocer objetos que no hayan visto con anterioridad, los objetos que pueden reconocer están limitados por las clases para las que han sido entrenados y les afecta en gran medida la distancia a los objetos y los cambios en las condiciones lumínicas, sombras y rotaciones. Esta dificultad aumenta también con el número de clases que la red sea capaz de detectar.

Algo también a tener en cuenta es el tiempo que tarde la red en realizar los cálculos necesarios para detectar el objeto y determinar su posición, algo fundamental en algoritmos que deben implementarse en tiempo real, ya que si tarda demasiado puede que la información ya no sea válida.

En las más de dos décadas de revolución de la detección de objetos hay dos etapas claramente diferenciadas, el periodo de detección de objetos tradicional (antes de 2014), y el periodo de detección basado en *deep learning* (después de 2014), como podemos ver en la Figura 2.2.

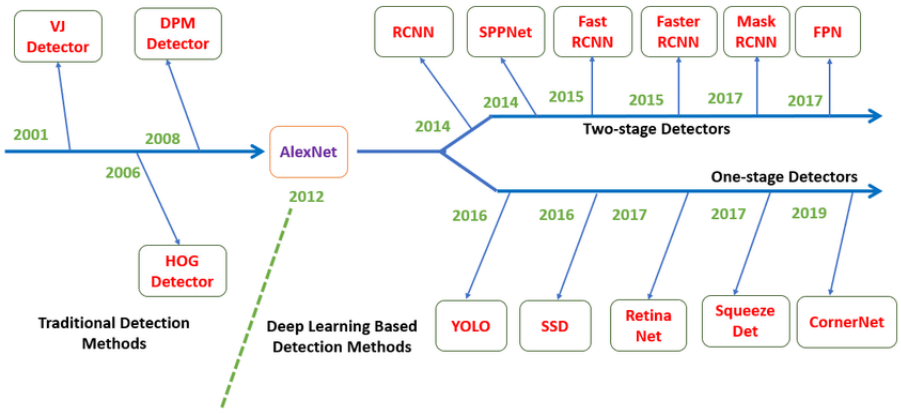


Figura 2.2 Cronograma de la tecnología de detección de objetos[2].

2.1.1 Detectores tradicionales

Si pensamos hoy en día de la detección de objetos es difícil separarla del *Machine Learning*, pero, debido a la falta de tecnología y de representaciones de imágenes en los primeros años, las personas que querían desarrollar este ámbito debían diseñar características de representación sofisticadas, y llevar al límite los reducidos recursos de computación de los que disponían.

- **Detectores de Viola-Jones**

  Fue en 2001 cuando Paul Viola y Michael Jones consiguieron llevar a cabo un algoritmo de detección de rostros humanos. Se trataba de un algoritmo en tiempo real rápido (cientos de veces más que cualquiera desarrollado hasta la fecha) y robusto, con gran tasa de acierto, en el que no influía aspectos como el color de la piel, ya que era procesado en escala de grises.

[1] Fuente: *https://www.researchgate.net/figure/The-increasing-number-of-publications-in-object-detection-from-1998-to-2018-Data-from_fig1_333077580* , [accedido 1 Jun, 2021]

[2] Fuente: *https://www.researchgate.net/figure/Milestones-of-object-detection-In-2012-the-major-turning-point-was-the-use-of-DCNN_fig1_341099304*, [accedido 1 Jun, 2021]

El algoritmo tenía cuatro etapas principales. En primer lugar, se usaba un filtro de Haar para detectar características comunes de las caras humanas, como pueden ser regiones más oscuras en la zona de los ojos que en la parte superior de las mejillas, o regiones más brillantes en el puente de la nariz, como podemos apreciar en la Figura 2.3 [28]. Además, la localización de ojos, nariz, boca, etc. también crean gradientes orientados de la intensidad de los píxeles.

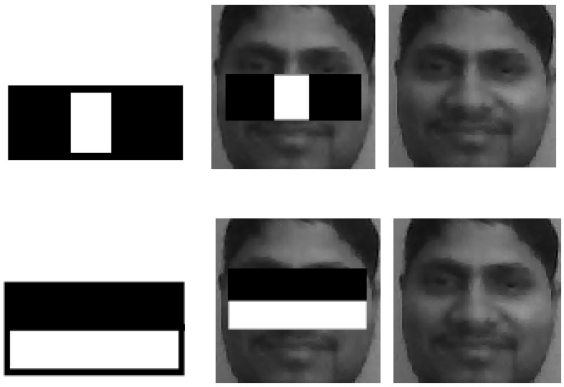


Figura 2.3 Características Haar aplicadas en rostros [28].

En segundo lugar, el algoritmo trabaja con la imagen integral, que se trataba de un método para acelerar el proceso de convolución, y que no usaba la imagen captada directamente, sino una representación de ella. En tercer lugar, se utiliza un algoritmo *AdaBoost*, para seleccionar una serie de características útiles en reconocimiento facial, dividiendo la imagen en secciones más pequeñas para después pasarlas por el elemento a continuación [19]. Este consiste en un detector multietapa, conocido como detector en cascada, que reduce el coste computacional focalizándose en la cara y no en los elementos del fondo, consiguiendo así una respuesta mucho más rápida [28].

- **Detector HOG**

  El Histograma de Gradiente Orientados o *Histogram of Oriented Gradients* (HOG) fue propuesto en 2005 por Navneet Dalal y Bill Triggs, diseñado principalmente para resolver el problema de reconocimiento de peatones. Su funcionamiento se basa en que la forma y la apariencia de los objetos en una imagen pueden ser descritos por la distribución de sus gradientes de intensidad, dividiendo la imagen en regiones más pequeñas llamadas celdas, buscando en ellas este tipo de gradientes [26].

  Puede ser capaz de detectar objetos de diferentes tamaños redimensionando la imagen de entrada en múltiples ocasiones, siendo un algoritmo base para muchos otros detectores, y se mantuvo en grandes aplicaciones durante años.

- **Modelo de partes deformables**

  El Modelo de Partes Deformables o *Deformable Part-based Model* (DPM) fue la cumbre de los métodos de detección de objetos tradicionales. Fue presentado en 2008 por Pedro Felzenszwalb como una extensión para el detector HOG, basándose, como podemos apreciar en la Figura 2.4, en entrenar considerando el aprendizaje como una forma de de descomponer un objeto en diferentes partes más sencillas para poder ser identificado en su totalidad [68].

Figura 2.4 Ejemplo visual de la descomposición de un objeto con DPM[3].

2.1.2 Detectores basados en *deep learning*

Entre 2010 y 2012 el progreso de los modelos de detección de objetos fue lento y sin grandes mejorías respecto a lo ya existente, hasta que en septiembre de 2012 la red AlexNet, creada por Alex Krizhevsky, compitió en el *Large Scale Visual Recognition Challenge* de ImageNet, ganando la competencia con un error top-5 del 15.3 %, más de 10.8 puntos por debajo del segundo competidor [25].

AlexNet

Consistía en una red convolucional de ocho capas: cinco capas convolucionales y tres totalmente conectadas, pero no era esto lo que la hacía única hasta la fecha, tenía una serie de características innovadoras [65]:

- **Múltiples GPUs.** La red AlexNet requería elevados costes computacionales, pero implementó por primera vez el uso simultáneo de varias unidades gráficas de procesamiento, consiguiendo entrenar modelos mucho mayores y reduciendo drásticamente los tiempos de entrenamiento.
- **Función ReLu.** En lugar de usar la tangente hiperbólica como función de activación, que era la norma por esa fecha para entrenar redes neuronales, la red de Krizhevsky implementó como función de activación la unidad lineal rectificada (ReLU), que consigue en determinadas pruebas alcanzar un 25 % de error hasta seis veces antes que usando la tangente hiperbólica, alcanzando un tiempo de entrenamiento mucho más reducido.
- ***Overlapping Max Pooling.*** Normalmente se usan las capas de *pooling* para poder reducir las dimensiones de los tensores, manteniendo la profundidad de la red intacta. La diferencia del usado en AlexNet respecto al *max pooling* común es que las ventanas usadas en este tipo de capas se superponen entre las adyacentes, realizando un filtrado con ventanas de 3x3 de tamaño, con un paso de 2 entre ventanas contiguas, y quedándose con el número máximo contenido en ellas. Si se comparara con ventanas no superpuestas de tamaño 2x2, las de 3x3 superpuestas consiguen reducir el error top-5 en un 0.3 % respecto a las otras ventanas [16].

[3] Fuente: *http://www.embeddedvisionsystems.it/images/automotive/dpm_bicycle.jpg* [accedido 2 Jun, 2021]

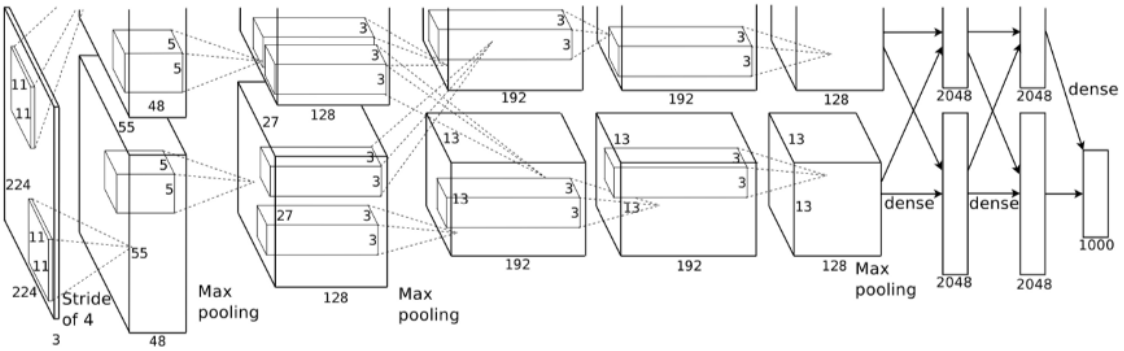


Figura 2.5 Arquitectura de AlexNet, pudiendo observar las capas de *max pooling*[4].

Algo importante a tener en cuenta, debido al gran número de parámetros de la red, 60 millones, es el conocido como *overfitting*. El sobreajuste se produce cuando un red neuronal se entrena tanto con el *dataset* que termina memorizando los ejemplos del entrenamiento en lugar de ser capaz de asimilar el concepto para aplicarlo en otras situaciones, por lo que no será capaz de trabajar bien en datos no vistos previamente. Alex solucionó este problema con el uso de dos métodos diferentes:

- ***Data Augmentation.*** Esta técnica consiste en obtener nuevas imágenes de entrenamiento modificando otras imágenes originales, creando así un *dataset* más variado. El resultado ofrecido es una mejora en las habilidades del modelo, forzando a la red para que use la lógica y en lugar de memorizar. Krizhevsky consiguió reducir la tasa de error top-1 en un 1 %, aplicando los trucos del *data augmentation*. Algunos de ellos pueden ser tan simples como hacer un *mirror* o espejo de la imagen original, duplicando así rápidamente las imágenes de entrenamiento. Otro método importante aplicado fue la traslación, o recortes aleatorios de la imagen, como podemos apreciar en la Figura 2.7, así como la anterior técnica en la Figura 2.6. Estas pequeñas variaciones en las imágenes sirven para enseñar a la red que, aunque los píxeles estén desplazados o invertidos, la fotografía sigue siendo la de un gato, en este ejemplo [16].

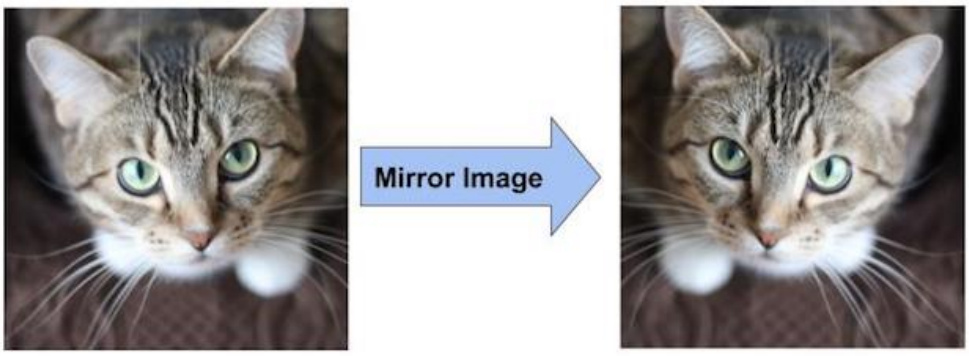


Figura 2.6 Ejemplos visuales de *data augmentation* [16].

[4] Fuente: *https://miro.medium.com/max/875/1*5bnqbGcBSLzaNMsz5dHkfg.png* [accedido 7 Jun, 2021]

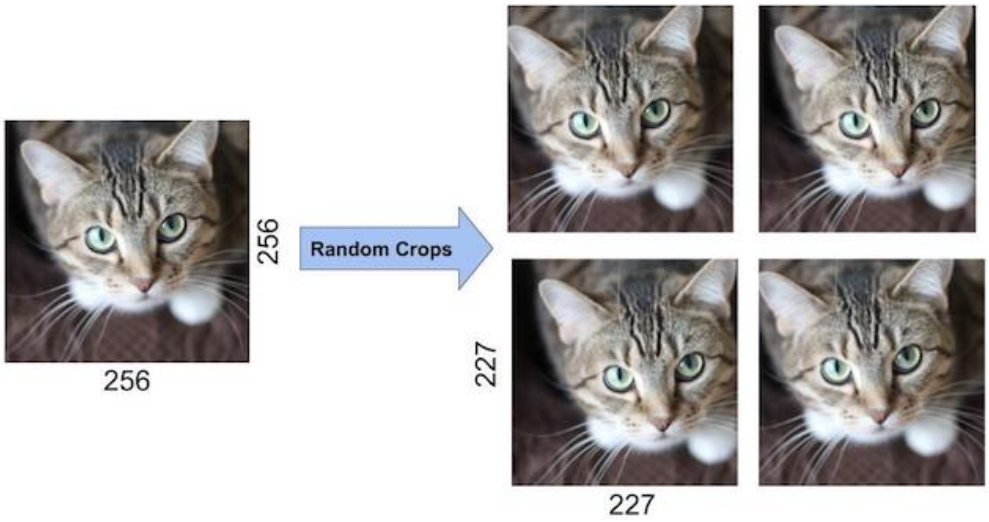


Figura 2.7 Ejemplos visuales de *data augmentation* [16].

Por último, un método algo más técnico fue aplicar el *Principle Component Analysis* (PCA), que consistía en cambiar las intensidades de los canales RGB. Todo esto incrementó en un factor de 2048 el número de imágenes del banco de datos [65].

- ***Dropout.*** Esta técnica consiste en "apagar" neuronas con una probabilidad preestablecida, por ejemplo 50 %. Con esto se consigue que en cada iteración se usen muestras diferentes de los parámetros del modelo, forzando la obtención de neuronas con características más robustas que puedan ser usadas con otras neuronas aleatorias. La contrapartida de esta técnica es que son necesarias más iteraciones para alcanzar la convergencia del modelo, lo que hace que el entrenamiento sea más largo.

Tras la creación de AlexNet, las siguientes evoluciones en la tecnología de detección de objetos basados en redes neuronales convolucionales vinieron divididas en dos ramas principales: los detectores de una etapa y los de dos etapas.

La diferencia fundamental entre ambos es que en los detectores de dos etapas se usa una red de regiones propuestas (*Region Proposal Network*) en la primera etapa, enviando estas propuestas a una segunda etapa clasificadora que detecta en estas regiones de interés si hay o no un objeto para el que ha sido entrenada. Tras la clasificación, se usa posprocesado para eliminar detecciones duplicadas así como para refinar las *bounding boxes* de los elementos identificados.

En cambio, los detectores de una sola etapa tratan la detección de objetos como un problema de regresión, en el que se recibe una imagen de entrada y se calculan los porcentajes de probabilidad de que una clase estudiada se encuentre en la imagen, así como el *bounding box* que recuadre el objeto. Esto hace a los detectores de una sola etapa sean considerablemente más rápidos, pero menos fiables ya que miran toda la imagen a la hora de encontrar objetos, en lugar de delimitar de forma anticipada las regiones de interés, obteniéndose por lo tanto una tasa de precisión bastante más alta y correcta, en general, en los clasificadores de dos etapas [61].

Two Stage Detectors

Uno de los casos más conocidos de este tipo de detectores es **R-CNN** (*Regions with Convolutional Neural Networks features*), el cual comienza con una serie de regiones propuestas en las que es posible que haya objetos. Cada una de estas regiones es reescalada hasta una dimensión de 227×227,

con la cual entra a la red de detección CNN entrenada con ImageNet para extraer características. Su salida, por cada región propuesta, es un vector de dimensión 4096, los cual pasan por clasificadores SVM (*Support Vector Machine*) lineales que son usados para predecir la presencia de un objeto dentro de cada región y para reconocer clases [38].

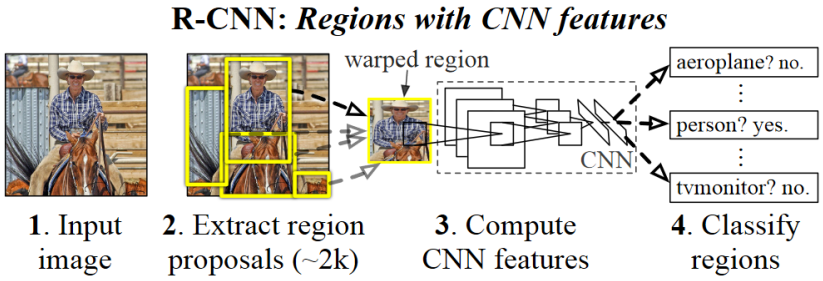


Figura 2.8 Esquema de la red R-CNN, mostrando los cuatro pasos que se ejecutan. (Fuente: *Paper* oficial R-CNN)

Esto conseguía una alta precisión del 58.5 % mAP con el *dataset* de PASCAL *Visual Object Classes Challenge 2007* (VOC07). Como se verá en el capítulo de Resultados, el *mean average Precision* (maP) es un cálculo que se usa popularmente para determinar la precisión media de los detectores de objetos. Pese a sus buenos resultados en precisión, las numerosas y redundantes extracciones de regiones propuestas (alrededor de 2.000 por cada imagen) producían que la velocidad fuera extremadamente lenta, tardando alrededor de 14 segundos en analizar una sola fotografía mediante el uso de la GPU. Pero poco después, en el mismo año, fue presentada **SPPNet** que vino a solucionar esta problemática [68].

Creada por Kaiming He y su equipo, *Spatial Pyramid Pooling Networks* (SPPNet) introduce la importante novedad de una capa espacial de *pooling* piramidal que permite que posteriormente una red neuronal convolucional pueda analizar cualquier imagen independientemente de su tamaño, sin la necesidad de reescalar a dimensiones concretas como 224×224, lo que limitaba la relación de aspecto y la escala de la imagen. Cuando las imágenes no eran de este tamaño era necesario recortar la imagen o estirarla, lo que producía que, con el recorte, el objeto quedara fuera de la imagen, o, con el estiramiento, que se produjeran distorsiones geométricas no deseadas en la imagen, lo cual podía provocar una disminución de la precisión de la red [39].

Figura 2.9 Ejemplos de recortes y deformaciones en imágenes para ser adaptadas a unas dimensiones preestablecidas. (Fuente: *Paper* oficial SPPNet)

Las CNNs constan de dos partes diferenciadas: las capas convolucionales y las capas totalmente conectadas. Es por estas últimas por lo que son necesarias entradas de dimensiones preestablecidas, debido a su propia definición. La solución fue la capa SPP ya mencionada para poder eliminar la restricción de las dimensiones preestablecidas. Después de cada capa convolucional fue agregada una

capa *spatial pyramid pooling* para que, cuando entrara esta salida en las capas totalmente conectadas, tuvieran unas dimensiones concretas. Básicamente lo que realizaban era agregar información extra a la imagen para no necesitar recortarla o estirarla.

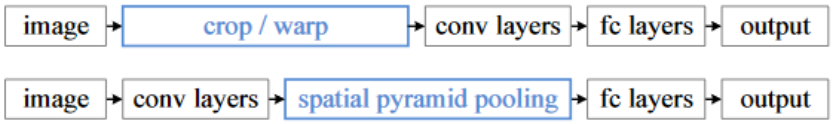


Figura 2.10 Metodología de SPP, abajo, frente a la metodología anterior de recortar/estirar la imagen, arriba. (Fuente: *Paper* oficial SPPNet)

En la Figura 2.10 podemos apreciar la metodología mencionada, introduciendo la capa SPP entre la convolucional y la *fully connected* en el diagrama inferior, mientras que en el superior se usar el recorte/estiramiento antes de la capa convolucional, produciendo deformaciones y pérdida de información.

Esta nueva metodología permite extraer el mapa de características de toda la imagen con un sólo procesado, sin necesidad de repetir continuamente en busca de rasgos, haciendo que SPPNet sea hasta 20 veces más rápida que R-CNN sin sacrificar la precisión, teniendo un mAP del 59.2 % con el dataset VOC07 [68].

Pero a pesar de esta gran mejora, SPPNet seguía teniendo ciertas deficiencias graves, las dos principales eran que el entrenamiento seguía siendo multietapa, y que durante este sólo se realizaba *fine-tuning* en las capas totalmente conectadas, ignorando todas las capas previas.

Para solucionar esta problemática llegó en 2015 **Fast R-CNN** de la mano de Ross Girshick, realizando una mejora de las dos redes mencionadas. Permitía entrenar de forma simultánea un detector y un generador de *bounding boxes* sin cambiar la configuración de la red para ello. Esto permitió aumentar la precisión de la red desde los 58.5 % de mAP con R-CNN hasta el 70.0 %, con una velocidad hasta 200 veces más rápida que R-CNN. Sin embargo, incluso con esta mejora en la rapidez, la velocidad todavía se veía limitada por el uso de regiones propuestas.

Poco después, también en 2015, fue presentada Faster R-CNN por Shaoqing Ren y su equipo, con la intención de solucionar el problema que lastraba la velocidad de este tipo de redes, remediando el cuello de botella que producía el uso de regiones propuestas, y convirtiéndose en el primer detector basado en *deep learning* que se ejecutaba casi en tiempo real. Para ello se introducía una red convolucional capaz de calcular estas regiones con coste computacional muy bajo, llamada *Region Proposal Network* (RPN) [68].

El sistema completo consta de dos módulos, el nuevo RPN y el Fast R-CNN, convirtiéndose en una sola red que completa el cometido de detcción de objetos, y que comparten los pesos de las capas convolucionales [38]. Desde R-CNN hasta Faster R-CNN, progresivamente se han ido integrando y unificando en un mismo marco de trabajo los bloques más independientes, como eran la detección de *proposals*, la regresión de *bounding boxes* o la extracción de rasgos [68]. Vemos en la Figura 2.11 cómo con RPN es posible generar regiones propuestas directamente a partir del mapa de características, permitiendo prescindir de un método externo para ello que consuma más recursos y necesite más tiempo de procesado.

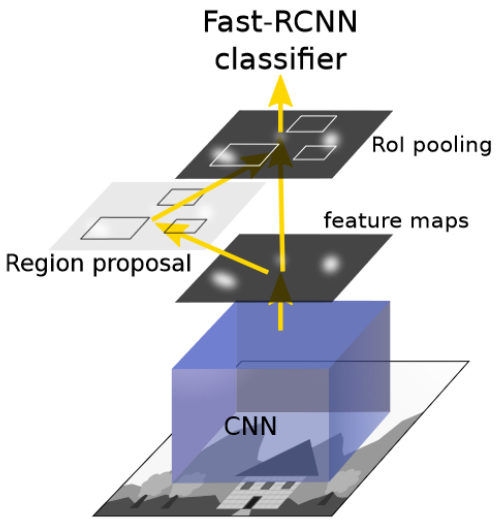


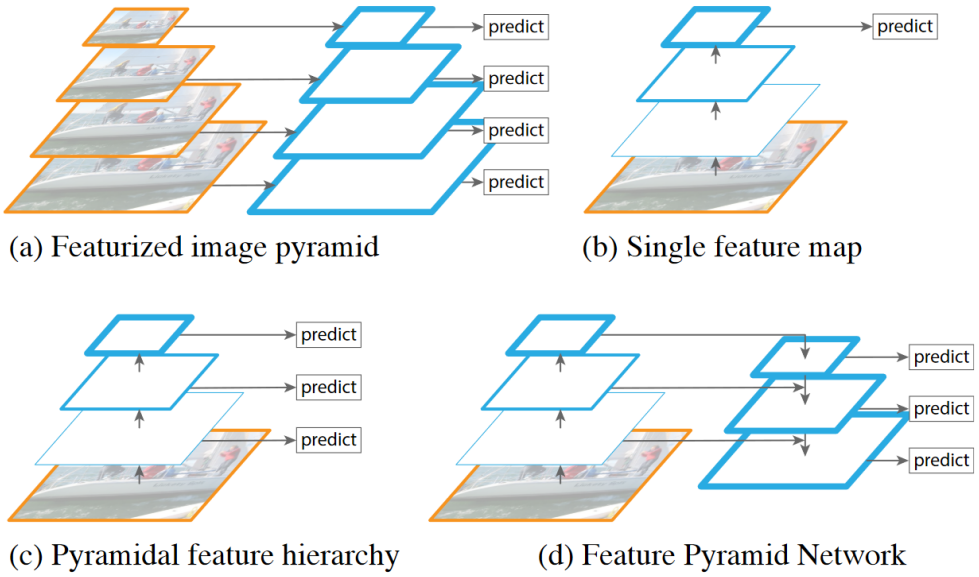
Figura 2.11 Esquema visual del procedimiento seguido por Faster R-CNN para la detección de objetos. (Fuente: *Paper* oficial Faster R-CNN)

Para implementar esta metodología se hizo uso de la arquitectura de red ZF-Net, con la cual se llegó a los 17 *fps* (*frames per second*), con una mAP del 73.2 % usando el *dataset* VOC07. Esto dejaba claro que el importante estrangulamiento que se producía en la red estaba solucionado, pero aún era posible enfrentarse a ciertas redundancias que tenían lugar durante etapas posteriores de la detección, por lo que en los años siguientes fueron propuestas ciertas mejoras, presentes en R-FCN y Light-Head R-CNN.

En 2017 Tsung-Yi Lin y su grupo de desarrollo presentaron *Feature Pyramid Networks* (FPN), basado en Faster R-CNN. La mayoría de los detectores basados en *deep learning* realizaban la detección únicamente en la capa final y, aunque las capas profundas de una CNN son buenas para reconocer clases en las imágenes, no lo son para la localización de objetos. Es por ello que con FPN se propuso una arquitectura *top-down* con conexiones laterales para contruir una mapas de característica de alto nivel en todas las escalas [68].

Aunque las pirámides de características son un componente básico en sistemas de reconocimiento para encontrar objetos en diferentes escalas, los detectores de objetos los han estado evitando debido a su alto coste computacional. Por esta razón *Feature Pyramid Networks* venía a proponer una jerarquía piramidal multiescala de redes convolucionales para construir pirámides de características con un coste computacional muy reducido [41].

En la Figura 2.12 se pueden apreciar cuatro métodos distintos para extraer mapas de características de las imágenes. En el primer caso (a), se usa una pirámide de imágenes para extraer una pirámides de *features*, lo cual es un proceso bastante lento, ya que las características se calculan en cada una de las escalas de la imagen de forma independiente. El segundo caso (b), es el que se estaba imponiendo en los sistemas de detección de objetos de la fecha, usando un mapa de una única escala para conseguir detecciones más rápidas. El tecer método (c), era una alternativa que consistía en reutilizar la jerarquía piramidal de características obtenida por una red convolucional como si fueran una pirámide de imágenes de características. Por último (d), se trata de la propuesta de Tsung-Yi Lin, siendo rápida como (b) y (c) pero más precisa, mostrando grandes avances en la detección de objetos con gran variedad de escalas. En todas las imágenes de la figura, las delimitaciones azules de los mapas de características son más gruesos cuanto más fuertes sean las características semánticas [41].



(a) Featurized image pyramid (b) Single feature map

(c) Pyramidal feature hierarchy (d) Feature Pyramid Network

Figura 2.12 Comparación técnicas usadas en detectores de objetos para extraer *geatures* de imágenes. (Fuente: *Paper* oficial *Feature Pyramid Networks*)

Pruebas con el dataset de MSCOCO, usando FPN en un sistema básico Faster R-CNN, muestran un $mAP@.5 = 59.1\,\%$ y un $mAP@[.5,.95] = 36.2\,\%$, convirtiéndose así en un estándar en muchos detectores de objetos posteriores [68].

One Stage Detectors

Los detectores de una etapa vinieron para ganar ampliamente en velocidad a los de dos etapas, siendo YOLO (*You Only Look Once*) el primer detector de una etapa en el periodo del *deep learning*, presentado por Joseph Redmon en 2016. Al ser el elegido para la realización de este proyecto nos centraremos especialmente en él en el apartado 3.5, así como en sus características de diseño, versiones, y en su especial forma de procesar las imágenes.

Por otra parte, *Single Shot MultiBox Detector* (SSD), presentado por Wei Liu en 2015, fue el segundo detector de objetos de una etapa en la época del *deep learning*. Es capaz de discretizar el espacio de salida por medio de *bounding boxes* predeterminadas de distintas relaciones de aspectos, formas y ubicaciones. Para predecir los objetos se generan puntuaciones de cada clase estudiada para cada una de estas cajas predeterminadas, y se realizan correcciones en las *bounding boxes* para que se ajusten mejor a la forma del objeto [43].

Las principales incorporaciones que introduce SSD son las técnicas de detección multi-resolución y multi-referencia, que incrementan la precisión y el acierto en gran medida con respecto a los detectores de una etapa de la época, sobre todo para objetos pequeños. La diferencia principal entre este algoritmo de detección y otros lanzados con anterioridad es que este es capaz de detectar objetos de diferentes tamaños en diferentes capas de la red neuronal, mientras otros, como la primera versión de YOLO, sólo lo hacían en la capa final de la red [68].

SSD es un algoritmo fácil de entrenar y de incorporar en sistemas que requieran un detector de objetos. Muestra de su fortaleza puede verse apreciada en diferentes pruebas que han sido realizadas con él, por ejemplo con imágenes del dataset VOC2007. El detector de Wei Liu y compañía es capaz de conseguir una precisión de $74.3\,\%$ de mAP con imágenes de entrada de 300×300 a una imponente tasa de 59 *fps* corriendo en una Nvidia Titan X, mientras que en imágenes de 512×512 el acierto aumenta hasta $76.9\,\%$ de mAP, alcanzando una mayor precisión que otros detectores *one*

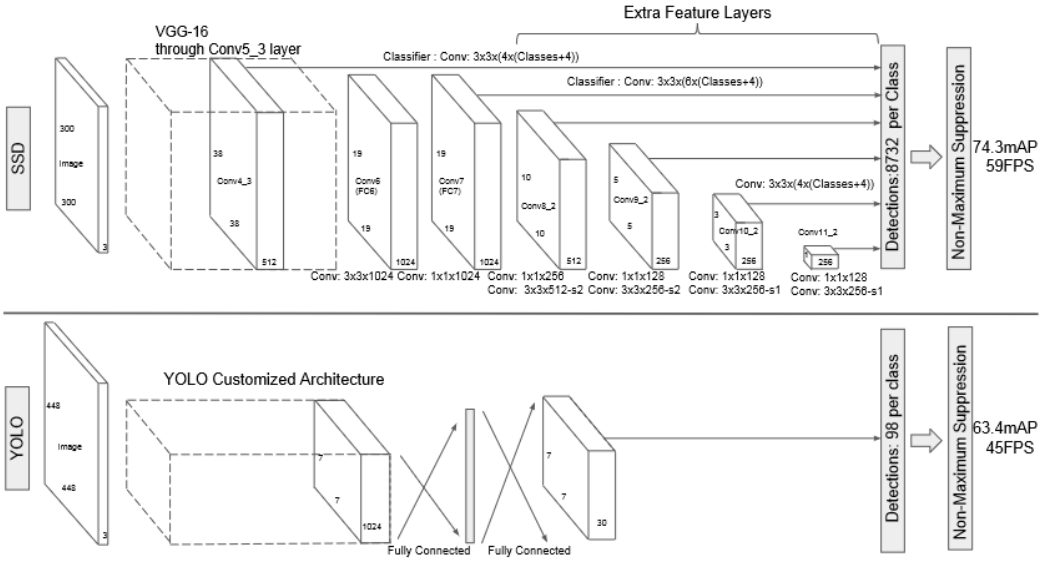


Figura 2.13 Comparación de la arquitectura de los dos modelos de detección mencionados hasta ahora: SSD vs. YOLO [43].

stage. Esta cifra consiguió superarse posteriormente en siguientes experimentos con técnicas de *data augmentation* hasta un 77.2 % mAP en 300×300 y 79.8 % mAP en 512×512 [43].

Como alternativa a estos dos detectores de una etapa surgió **RetinaNet**, que venía a solucionar la derrota que sufrían los detectores de una etapa frente a los de dos etapas en términos de precisión en las predicciones, ya que los segundos se basan en regiones de interés propuestas que consiguen mejores predicciones. Sus creadores, Tsung-Yi Lin y su grupo, llevaron a cabo una investigación acerca de por qué se producía esta gran diferencia, sacando la conclusión de que se debía principalmente a la problemática conocida como *Class Imbalance*. Este desequilibrio entre clases ocurre cuando ciertas situaciones de salida son mucho más probable de darse que otras. En el caso concreto de detección de objetos, los detectores de una etapa pueden evaluar entre 10^4 y 10^5 posibles localizaciones en la imagen, pero sólo unas pocas contienen de verdad objetos. Esto da lugar a dos problemas fundamentales: primero, el entrenamiento es ineficiente, ya que el elevado número de casos negativos hace que lo aprendido no sea apenas útil, y segundo, esta exagerada cifra de negativos pueden conducir a un deterioro del modelo [42].

Para solucionar esto, los desarrolladores de RetinaNet introdujeron una nueva función *loss*, denominada *focal loss*, reformulando así la forma en la que el modelo era entrenado, y haciendo que se prestara más atención a los ejemplos difíciles o mal clasificados. La idea era que si durante el entrenamiento se daban casos difíciles de falsos positivos en los que la red fallaba, poder añadir esas muestras concretas al conjunto de entrenamiento para que, cuando el clasificador volviera a entrenarse, poder actuar mejor con ese nuevo conocimiento. Es esto a lo que se conoce como *hard negative mining*.

Focal loss llevó a este detector de una etapa a alcanzar precisiones comparables a las de detectores de dos etapas, consiguiendo mantener altas tasas de velocidad características de los *one stage*. Ejemplo de ello son las pruebas realizadas usando el *dataset* COCO, con distintos niveles de velocidad vemos en la Figura 2.14 que tiene un tiempo de inferencia algo mayor, pero su precisión es también bastante más elevada que su competencia.

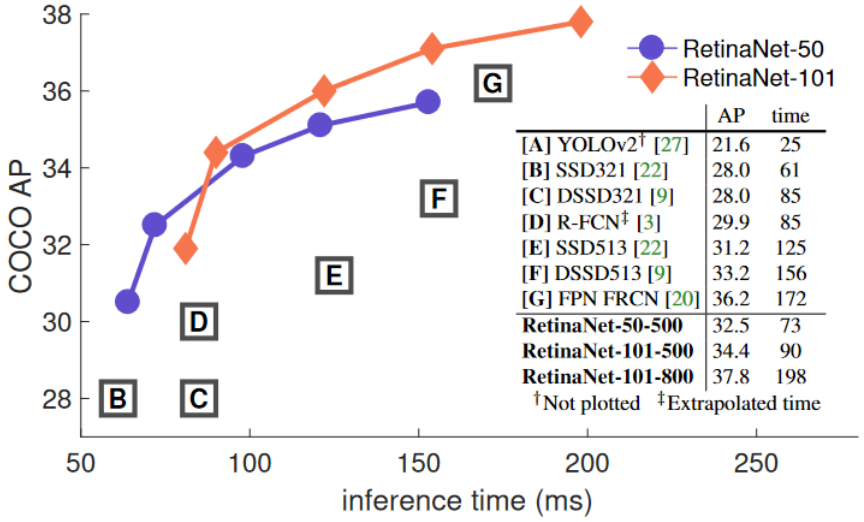


Figura 2.14 Comparación de precisión frente a velocidad de RetinaNet con otros detectores, usando el dataset de COCO. (Fuente: *Paper* oficial RetinaNet)

En diciembre de 2019 Xianzhi Du y su equipo, de la mano de Google, presentaron Spinet, una red que introducía una novedosa arquitectura nombrada como *scale-permuted*, la cual venía a sustituir la estructura piramidal alegando que esta no era efectiva para localizamiento y reconocimiento de manera simultánea, sino sólo para tareas de clasificación [33]. Con esta nueva metodología se alternan capas de convoluciones de diferentes tamaños por las que pasa la información de la imagen, consiguiendo una mejora con respecto a las redes piramidales.

En la Figura 2.15 se puede apreciar ejemplos de una comparación entre la estructura de red piramidal frente a la nueva propuesta de Spinet, en la que la anchura del bloque indica la resolución de la característica, y la altura indica la dimensión de la característica. Las líneas de puntos representan las conexiones desde/hacia los bloques no representados.

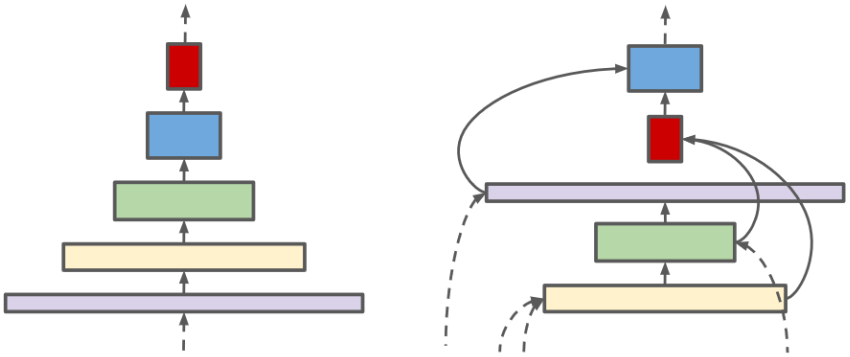


Figura 2.15 Ejemplo de red *scale-decreased* o piramidal, izquierda, frente a red *scale-permuted*, derecha [33].

2.2 Conducción Autónoma

Uno de las aplicaciones más ambiciosas de la inteligencia artificial es la conducción autónoma. Un vehículo autónomo es aquel que es capaz de dirigirse de forma independiente, sin ayuda humana, mediante el software, los sensores, procesadores y actuadores de los que este dispone. Normalmente suelen tener un destino prefijado, o un recorrido preestablecido, siendo capaces también de improvisar en determinadas circunstancias [9].

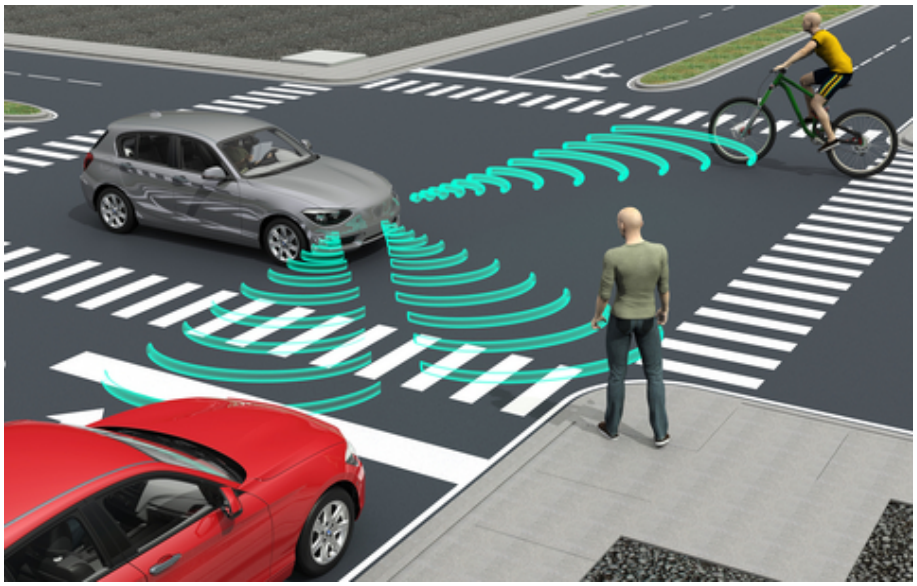


Figura 2.16 Concepto de vehículo autónomo en espacio urbano[5].

Los vehículos autónomos se cimientan en el *deep learning*, para poder aprender sobre la marcha nuevos estímulos, como pueden ser carreteras y caminos por donde circular, objetos, señales o situaciones adversas, pudiendo instruirse para circular, por ejemplo, por vías que no están destinadas para ello [15]. Cuando se habla de conducción autónoma suele dividirse en cinco grados, dependiendo de la mayor o menor influencia que tenga el ser humano sobre ella.

- **Nivel 0.** Sin automatización en la conducción.

  Es el modo convencional que lleva implantado desde el surgimiento de los vehículos a motor. Todos las tareas relacionadas con la conducción son realizadas por el conductor, el cual tiene el control absoluto del automóvil.

  Los coches algo más modernos que cuentan con asistente de mantenimiento de carril, frenado de automático en caso de emergencia u otros sistemas básicos de seguridad activa (ASS por sus siglas en inglés) también irían ubicados en este nivel, ya que únicamente se activan de forma momentánea si el conductor ha sufrido una distracción.

- **Nivel 1.** Asistencia en la conducción.

  En este nivel se incluyen aquellos vehículos que han experimentado una ligera evolución en la conducción. Se pueden incluir en esta escala aquellos automóviles con asistencia en la conducción que son capaces de controlar movimientos laterales o longitudinales, pero no ambos a la vez.

[5] Fuente: *https://e3sparkplugs.com/media/pcon/dreamstime_xs_87122115-socialmedia-1167.jpg*

Muestra de esta innovación puede ser, por ejemplo, el control de velocidad de crucero adaptativo. Este es un sistema que controla activamente la velocidad, no manteniendo únicamente la velocidad programada, sino siendo capaz de acelerar y frenar cuando sea oportuno, manteniendo la distancia de seguridad frontal con los demás vehículos del entorno. Son múltiples los fabricantes que incluyen ya esta ayuda en sus automóviles convencionales, como pueden ser el Ford Focus, Renault Megane, Volkswagen Polo o Toyota Prius entre otros.

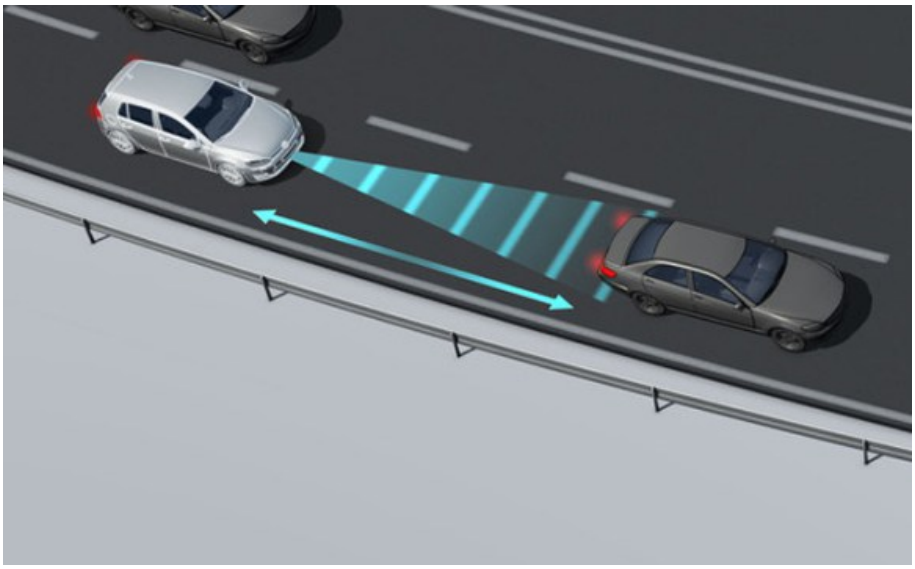


Figura 2.17 Concepto de control de velocidad de crucero adaptativo[6].

Otra muestra de este nivel sería el asistente de ayuda al aparcamiento, el cual únicamente actúa sobre la dirección del volante, siendo el conductor el que debe operar sobre los pedales de aceleración y freno.

En líneas generales son sistemas que conciben una conducción más cómoda pero que siguen requiriendo una concentración absoluta del conductor.

- **Nivel 2.** Automatización parcial de la conducción.

  En este nivel se encuentran los sistemas que cuentan con control del movimiento tanto lateral como lateral de forma simultánea. El conductor sigue teniendo la total responsabilidad del vehículo, ya que las situaciones en las que se puede usar son limitadas, y el vehículo no es capaz de dar un plan de respuesta antes situaciones imprevistas, como pueden ser obstáculos.

  Podría catalogarse como un piloto automático para autopistas, consiguiendo el mantenimiento del automóvil en el centro del carril trabajando de forma simultánea con el control de velocidad adaptativo. El primer coche comercial de nivel 2 fue el Mercedes-Benz Clase S 2013, posteriormente modelos de Tesla como el Model X. Hoy en día gran cantidad de automóviles pueden ser catalogados con este rango.

  El sistema de aparcamiento totalmente automático también se incluiría en este rango, ya que controla ambos movimientos sin necesidad del conductor, el cual sí debía operar sobre los pedales en la escala anterior.

- **Nivel 3.** Automatización condicionada de la conducción.

  A partir de este nivel el conductor puede solicitar que el vehículo tome el control total de la

[6] Fuente: *https://www.ptcarretera.es/wp-content/uploads/2018/07/control-de-crucero-adaptativo.jpg*

conducción, dentro de ciertas limitaciones. No es necesario que se supervise la conducción, pero sí debe estar alerta por si el sistema lo requiera ante una situación de riesgo ante la que no sepa cómo actuar, o cuando se encuentre en límites para los que no esté diseñado, como puede ser velocidad excesiva o líneas que delimitan los carriles de forma confusa.

El vehículo también será capaz de analizar el entorno para, en caso de emergencia o despiste del conductor, realizar la maniobra más adecuada para evitar accidentes. Es una muy buena alternativa en situaciones repetitivas, como manejo en atascos o tráfico denso [3].

Figura 2.18 Prototipo de Ford Fusion Hybrid para pruebas de conducción autónoma[7].

Unos de los grandes impedimentos a los que se enfrentan este tipo de automóviles es la legislación actual, la cual, a día de hoy, aún no ha contemplado este tipo de automatizaciones en la mayoría de los casos. Países como Alemania quieren dar un paso al frente para legislar esta tecnología, quien recientemente ha presentado un borrador de la ley [5], buscando también un consenso entre países, ya no sólo de nivel 3 de automatización, sino también los niveles 4 y 5.

- **Nivel 4.** Automatización elevada de la conducción.

  En este escalón es conductor prácticamente es innecesario, salvo en contadas ocasiones. El vehículo es capaz de estar al mando de la conducción de forma constante, sin la necesidad de que el conductor esté alerta en caso de emergencia, ya que está programado para actuar frente a situaciones imprevistas y hacerlo de la manera más segura (situación de mínimo riesgo).

  El conductor podría tomar el control en algún momento si quisiera, siendo el coche el que decidiera el momento para el cambio en el momento oportuno, pero sólo sería realmente necesario cuando se llegue a un ámbito para el que no esté programado. En este caso, además, el tiempo en el que el conductor debe retomar el control del vehículo puede ser de varios minutos, frente a los pocos segundos en los que debía reaccionar en el nivel 3. En caso de que el conductor no retomara la conducción el vehículo sería capaz de posicionarse en una zona segura de forma automática.

  En este nivel no se encuentran aún coches a la venta, solamente distintos prototipos de varias

[7] Fuente: *https://static.motor.es/fotos-noticias/2017/02/min652x435/ford-vehiculo-autonomo-201734034_1.jpg*

marcas. Una de las representaciones más conocidas de este escalón es Waymo, el coche autónomo de Google. La tecnología que incorpora el coche realiza una lectura constante del entorno, transportando a los ocupantes hasta el destino seleccionado y eligiendo la ruta más oportuna para ello.

Un aspecto importante que ayudaría mucho en la incursión de este tipo de vehículos sería la posibilidad de comunicarse con otros coches autónomos y con el entorno, como podría ser con semáforos o cámaras externas. Ayudaría en gran medida a aumentar el tiempo de respuesta, así como la seguridad y la estabilidad.

Figura 2.19 Waymo, coche autónomo de Google[8].

- **Nivel 5.** Automatización completa de la conducción.

  Este sería el nivel máximo de automatización, mediante el cual el vehículo es capaz de manejar en cualquier circunstancia en las que la haría un ser humano, careciendo incluso de volantes y pedales, y por consecuencia, de conductor. No tendría limitaciones climatológicas ni geográficas, por lo que serían necesarias nuevas medidas de localización y balizamiento en el entorno, siendo indispensables, por ejemplo, sistemas inalámbricos de comunicación vehículo a vehículo (V2V) y vehículo a infraestructura (V2I). Actualmente no hay ningún modelo a este nivel, solamente distintos prototipos en fases de prueba.

  Una de las dudas principales respecto a este tipo de vehículos es su funcionalidad en el medio rural [49]. En aplicaciones donde la conectividad a internet es algo esencial para su correcto funcionamiento, el mal acceso a este bien indispensable hoy en día puede suponer un grave peligro. Es ciertamente contradictorio cómo en pleno año 2021 se planteen alternativas como el coche autónomo pero sin embargo haya un problema real en el acceso a internet en zonas montañosas o numerosos pueblos rurales. Esto se debe principalmente a que las grandes compañías privadas de telefonía e internet no ven como una opción rentable el invertir en estas zonas, por lo que acaban teniendo una gran desventaja respecto a las ciudades en este ámbito. Es por ello que también se empiezan a estudiar coches autónomos que no dependan de internet para poder circular, como es el caso del coche de Google ya mencionado, siendo algo sorprendente en una compañía cimentada en la red.

  Algo también preocupante en estos proyectos son los ciberataques, que podrían, en principio,

[8] Fuente: *https://i.blogs.es/402b3b/google-car-2/1366_2000.jpg*

evitarse si el vehículo garantiza su funcionamiento de forma *offline*. Es un problema grave del que ninguna empresa está exenta, como se ha visto recientemente con Tesla. La importante compañía de vehículos eléctricos y autónomos se ha visto afectada por una violación de la seguridad que daba acceso a diversas cámaras de su fábrica en China [14]. A pesar de ello es una firma que tiene presume de su seguridad antes estos ataques, e incluso retó con casi un millón de euros de recompensa a hackers que fueran capaces de exponer vulnerabilidades en uno de sus modelos, el Tesla Model 3.

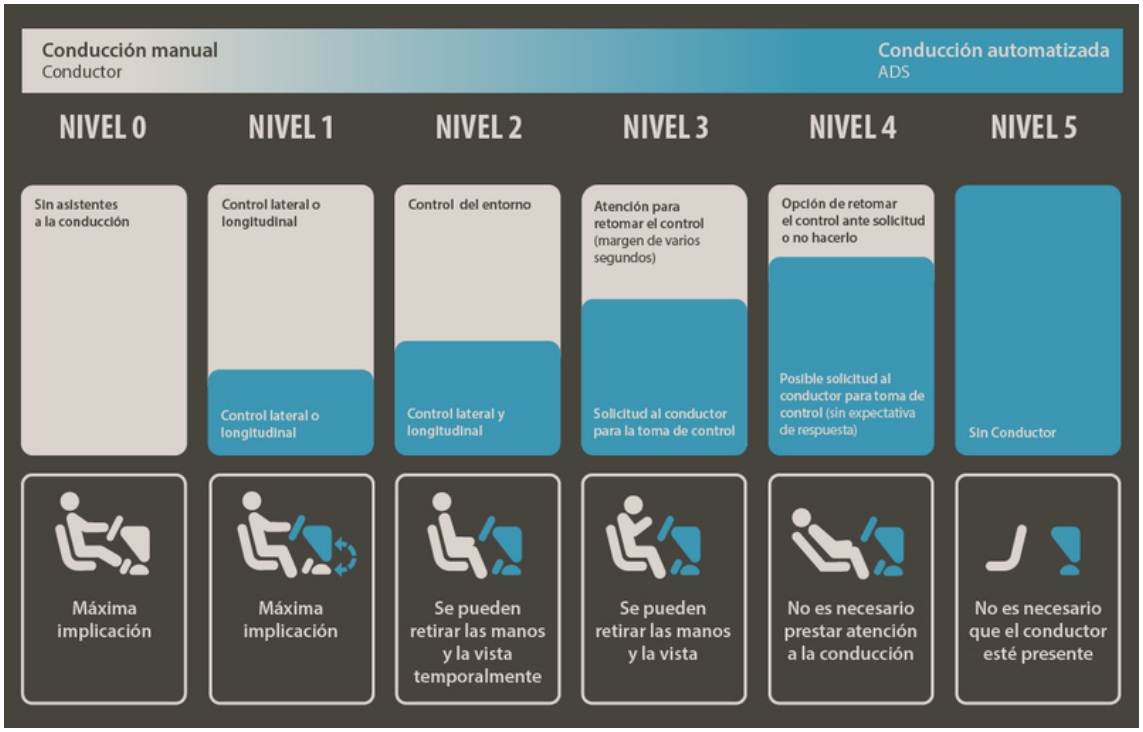


Figura 2.20 Esquema de los niveles de conducción autónoma[9].

Todo esta automatización es posible gracias a la cantidad de sensores y cámaras que equipan estos automóviles. Los sensores son los ojos de los vehículos autónomos, siendo las cámaras uno de los componentes vitales [2]. Con la información proporcionada y la utilización en visión de inteligencia artificial se pueden llegar a distinguir obstáculos, personas, ciclistas o animales, y su combinación con sensores LiDAR y radares pueden realizar mapeados 3D del entorno por donde está circulando. LiDAR es el acrónimo de *Laser imaging detection and ranging*, y como su nombre indica utiliza haces láser pulsados para calcular la distancia entre el emisor y un objeto que se encuentre en el rango de medida. La ventaja que representa el láser frente a los ultrasonidos convencionales es que la distancia de alcance y la velocidad es mucho mayor, pudiendo adquirir descripciones del entorno con mucho mayor detalle, pudiendo generar mapas en tres dimensiones basados en nubes de puntos. La elevada potencia de cálculo y la utilización de algoritmos requieren de un potente computador, ya que la toma de decisiones debe realizarse en un muy reducido espacio de tiempo, y puede ser una cuestión vital el tiempo de cómputo.

Con la revolución de vehículos autónomos que se avecina el conductor será siempre una pieza clave, y no debe olvidar su importante labor al volante. Debe tener siempre claro qué funciones es capaz de realizar el vehículo que está conduciendo, y atender a las limitaciones del mismo, debido a que durante varios años se estarán conduciendo coches con diferentes niveles de automatización.

[9] Fuente: *https://www.km77.com/images/medium/5/7/9/5/2autonomo-km77-3-.335795.jpg*

Uno de los principales peligros de los niveles bajos de automatización es que el conductor olvide o se despiste ante ciertas tareas esenciales creyendo que será el vehículo quien las realice, confiando más labores de las que realmente el sistema puede atender.

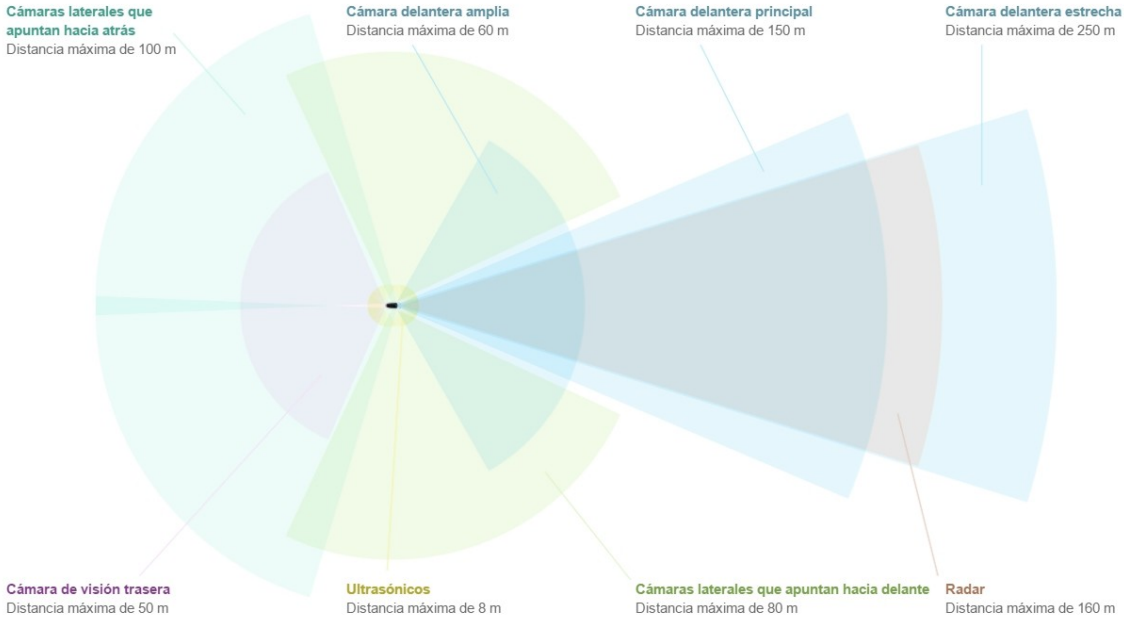


Figura 2.21 Sensores de un Tesla Model S, con Autopilot 2.0, y sus áreas de acción[10].

Además de los sensores de lectura del entorno también será necesario el afamado GPS y GLONASS, pero también otras innovaciones que se basan en ciudades inteligentes. Ejemplo de ello pueden ser los sensores de carretera (*Roadside sensors*), los cuales estarían instalados en las propias infraestructuras urbanas, siendo capaces de medir las condiciones del tráfico o incluso alertar de obstáculos como colisiones de tráficos, peatones, árboles caídos, desprendimientos de rocas, etc [4]. Los coches inteligentes serían capaces de manejar esta información para trazar rutas alternativas de antemano, o realizar maniobras seguras. Los coches también podrían mandar información a estos sensores, que a su vez podrían enviar los datos a centrales para gestionar cálculos que no necesiten ser instantáneos y liberar ancho de banda de abordo para tareas críticas más urgentes en tiempo real como la evitación de obstáculos.

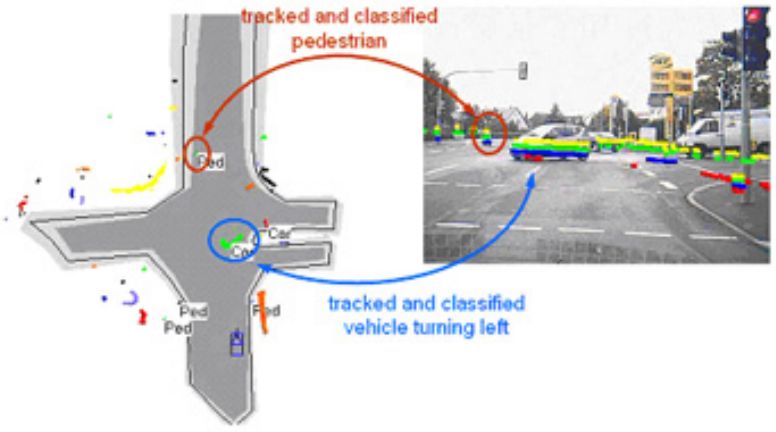


Figura 2.22 Ejemplo de uso de un sensor de carretera para detectar peatones en intersecciones peligrosas[11].

[10]Fuente: *https://i.blogs.es/8447f8/sensores-tesla-autopilot-2/1366_2000.jpg*
[11]Fuente: *http://www.safespot-eu.org/images/sp/sp2_1.jpg*

Estos sensores podrían ser en determinadas situaciones una ayuda vital, como reduciendo la velocidad de forma automática cuando la red de sensores detecta un viandante cruzando la carretera incluso antes de que lo detecten los sensores lídar del propio vehículo o sabiendo con anterioridad cuando los semáforos van a cambiar. En conclusión, hay infinidad de aplicaciones y usos que se pueden asignar a nuevos sensores dependiendo de la imaginación y predisposición que se tenga a esta tecnología de cara al futuro.

2.3 Proyectos similares

Este trabajo fue planteado con miras a la creación de un vehículo acuático autónomo capaz de detectar y clasificar obstáculos que se puedan ir observando a su alrededor.

Los barcos autónomos no están tan evolucionados como los coches de conducción autónoma, ya que no se ha apostado tanto por su desarrollo, pero se prevé que en los próximos años surjan numerosas innovaciones en este ámbito.

Es por ello que desde 2017 la Organización Marítima Internacional estudia normativas para garantizar que estos buques, designados como buques MASS (*Maritime Autonomous Surface Ship*), puedan operar de forma limpia y sin riesgos, siendo un requisito indispensable que sean, al menos, tan seguros como los tradicionales y mejoren la eficiencia [47]. Pero como en todas las grandes innovaciones hay detractores que consideran que estas embarcaciones sin personal a bordo destruirá miles de empleos en los casi 100.000 que navegan a diario transportando gran parte de toda la mercancía mundial.

Pese a ello, el sector no pretende estancarse en tecnologías pasadas, y se han creado proyectos como *Autoship* que tiene como objetivo aportar financiación para acelerar la transición hacia una nueva generación de buques autónomos en la Unión Europea, prometiendo establecer relaciones comerciales mediante buques autónomos para finales de 2023.

Figura 2.23 Imagen aérea del buque autónomo Yara Birkeland[12].

Uno de los barcos autónomos más destacables es el buque **Yara Birkeland**, que con la tecnología marítima de la compañía noruega Kongsberg fue diseñado para ser «*el primer portacontenedores*

[12]Fuente: *https://forococheselectricos.com/wp-content/uploads/2020/12/Yara-Birkeland-3.jpg*, [accedido 30 Jun, 2021]

totalmente autónomo y sin emisiones» [23]. Su desarrollo fue algo retrasado por el *COVID-19*, pero finalmente en se entregó a la empresa de fertilizantes también noruega Yara Norge AS, quien lo encargó en 2017. El buque tiene una capacidad de carga de 120 TEU, capaz de alcanzar una velocidad máxima de 13 nudos, siendo impulsado por dos vainas y dos propulsores que obtienen la electricidad de un enorme paquete de baterías de 7 MWh de capacidad [29].

Está pensado para que la carga y descarga de la mercancía se realice también de forma automática mediante grúas y demás maquinaria eléctrica, como se puede apreciar en la Figura 2.24, y cuenta con un sistema de amarre automático sin necesidad de infraestructuras especiales en el muelle ni intervención humana [29].

Figura 2.24 Imagen *renderizada* del Yara Birkeland durante un proceso de descarga [29].

Aunque actualmente se encuentra aún en periodo de pruebas, se estima que para final de 2021 será cuando empiece a transportar entre un 40-60 % de los contenedores de la compañía, y se está preparando para que pueda funcionar sin ningún tipo de tripulación dos años más tarde.

Otro barco con características similares que también opera en la costa noruega es el **Eidsvaag Pioneer**, el cual será automatizado mediante la ya mencionada iniciativa Autoship. Se trata de un barco de transporte de alimentos para la acuicultura que navega por toda la costa oeste del país.

Figura 2.25 Imagen del buque Eidsvaag Pioneer, seleccionado para el proyecto Autoship[13].

[13]Fuente: *https://www.vesselfinder.com/vessels/EIDSVAAG-PIONER-IMO-9660449-MMSI-258729000*, [accedido 30 Jun, 2021]

Uno de los mayores retos a los que se enfrentan sus desarrolladores es que se trata de una ruta marítima corta, pero con cambios de clima que pueden ser demasiado bruscos y provocar que la integridad del barco se comprometa, por lo que es una misión difícil y en la que se tiene que dedicar un gran trabajo.

Con proyectos de este tipo, la iniciativa Autoship quiere demostrar empíricamente que es posible operar, ya sea de forma automática o desde tierra, un gran número de barcos por todas las zonas geográficas, y que es un mercado con gran potencial [21].

Por último, una embarcación considerablemente más destacable en lo relacionado a este proyecto, la cual mantiene, además, unas dimensiones mucho más reducidas es el ***Mayflower Autonomous Ship*** (MAS), desarrollado por IBM y la organización de investigación marina sin ánimo de lucro ProMare, quien se enfoca en el conocimiento y protección de los océanos. Su nombre es dado en honor al galeón que en 1620 realizó el trayecto de los primeros ingleses en llegar a América del Norte a colonizar nuevas tierras [67]. Cuenta con unas dimensiones de 15 metros de largo y 6.2 de ancho, lo que lo convierte en una embarcación de 5 toneladas fabricada en aluminio, capaz de moverse a unos 10 nudos (alrededor de 18 km/h) [20].

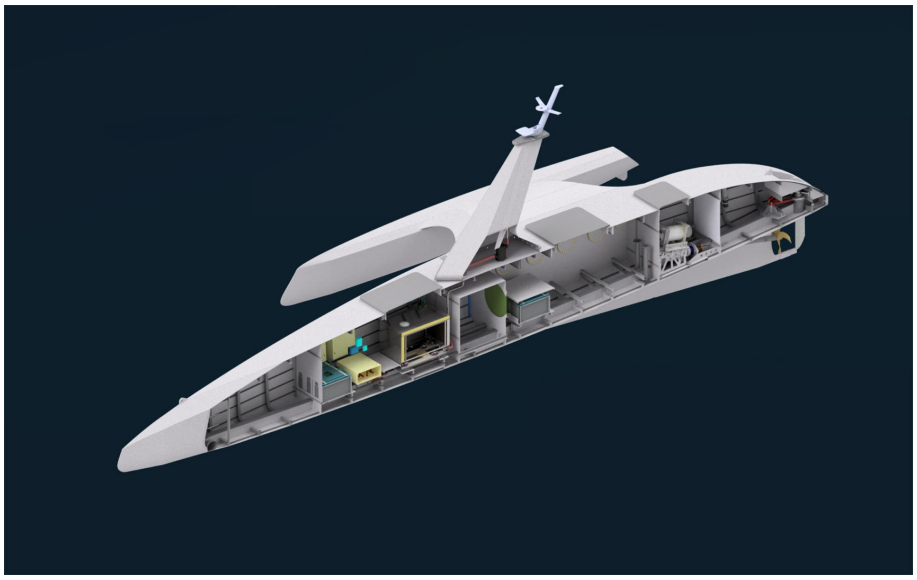


Figura 2.26 Sección *renderizada* del Mayflower, en el que se aprecia la ausencia de cabinas en su interior con el fin de centrar todo su compartimento en mecánica y funcionalidades tecnológicas [52].

El MAS es propulsado mediante dos motores de 4 y 22 KW en formato PVM, eliminando la necesidad de toma de agua y con ello los problemas de corrosión y fuga que pueden surgir. Aparte de estos, también dispone de otros dos motores de eje Easybox que funcionan con un potencia de 20kW a 600 revoluciones por minuto. Adquieren la energía necesaria mediante generadores eólicos y placas solares, consiguiendo así una enorme autonomía tanto cuando el día esté soleado como cuando haya viento. Esta energía eléctrica será almacenada en 8 baterías de iones de litio de 38,4 kWh, que puede llegar a proporcionar 800 Ah a 48V. Pese a ello, para asegurar la embarcación en caso de fallo en las otras fuentes de energía, también dispone de un pequeño generador diésel [52].

Como se ha señalado anteriormente, su aspecto más destacable es la ausencia de un capitán de la embarcación, o al menos es lo que puede parecer en un principio, ya que cuenta con un capitán constituido por una inteligencia artificial, la cual trabaja de manera similar a como lo haría un capitán humano: realiza constantemente correcciones sobre su ruta y su ritmo de navegación, asimila

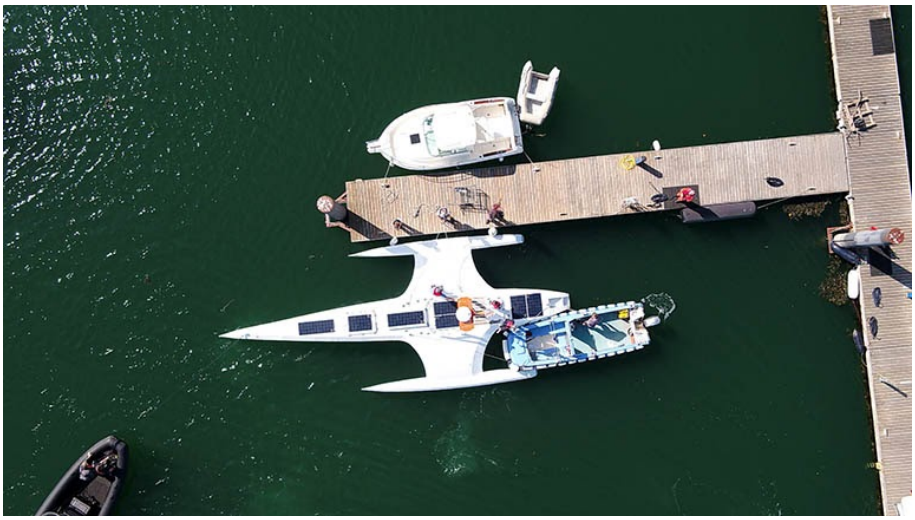


Figura 2.27 Vista de pájaro del *Mayflower Autonomous Ship* [20].

datos recibidos o calculados de distintas fuentes, y escanea el horizonte en busca de obstáculos o peligros, como barcos, boyas o rompeolas entre otros peligros. También cuenta con una conexión directa con fuentes de datos meteorológicos, lo cual le permite advertir con anticipación tormentas que puedan resultar peligrosas para la navegación o para la integridad de la embarcación, pudiendo ajustar así su posición y su inclinación, entre otros, para realizar su labor [67] [52].

La red de Inteligencia Artificial del Mayflower ha sido estando entrenada durante dos años por su equipo de desarrollo mediante potentes equipos IBM Power AC922, los cuales estaban compuestos por las CPU IBM Power9 y las GPU NVIDIA V100 Tensor Core, utilizando como *dataset* más de un millón de imágenes náuticas captadas por cámaras ubicadas en Plymouth Sound, en el Reino Unido, así como bancos de imágenes de acceso libre [37].

Todo esto no sería posible sin su elevado número de cámaras y sensores. El buque de diseño de trimarán futurista, orientado a ser lo más eficiente posible y a reducir al mínimo su rozamiento con el agua [52], cuenta con más de 30 sensores a bordo y 6 cámaras conectadas mediante la IA que gobierna la embarcación, que es capaz de ejecutarse sobre la base de 6 Nvidia Jetson AGX Xavier, 2 Nvidia Jetson Xavier NX, 4 ordenadores con base Intel y 4 sistemas de microprocesadores personalizados, lo que le permite tomar decisiones en tiempo real de forma rápida. Además, como no podía faltar en un barco, cuenta con un potente sistema de posicionado y navegación formado por equipos GNSS de precisión (Sistema Global de Navegación por Satélite), IMU (Unidades de Medición Inercial), radares, estación meteorológica, SATCOM y AIS [20].

Para conmemorar el trayecto del navío de 1620 que le da nombre, se planeó un viaje para la primavera de 2021 de 5.000 kilómetros que cruzara el Atlántico en apenas 20 días, desde el puerto de Plymouth, en Reino Unido, hasta el puerto del mismo nombre ubicado en Massachusetts, Estados Unidos. Además de la propia hazaña de conseguir este reto, el buque realizaría también estudios durante sus recorridos acuáticos sobre poblaciones de ballenas y vida marina, niveles de microplásticos y contaminación, salinidad, y la energía que pueden tener las olas en mar abierto, ayudando así a los científicos a estudiar mejor asuntos como el cambio climático o el declive en la salud de los océanos [52]. Esta ambiciosa hazaña no pudo completarse en un primer intento, y el barco tuvo que volver a puerto 3 días después de su partida por problemas mecánicos, pero en cuanto las reparaciones sean completadas, el viaje será retomado para completar su cometido [51].

La experiencia adquirida por el sector durante el desarrollo del buque servirá para promover la creación de nuevas naves comerciales autónomas, ayudando a transformar el futuro de la investigación marina y estableciéndose como un importante punto de partida para futuros proyectos similares [37]. Las condiciones en tiempo real del barco como su ubicación, velocidad, condiciones ambientales, los datos de sus diveros proyectos son monitorizadas y de libre acceso mediante su *página web oficial*, con la que también se pretende concienciar a la población sobre la protección de los mares. En ella también pueden encontrarse incluso en algunas ocasiones vídeos en directo de las cámaras del propio Mayflower [67].

Figura 2.28 Buque autónomo Mayflower durante uno de sus viajes [51].

De igual manera que ocurría en los vehículos autónomos terrestres, como se estudia en [18], en los barcos también se podrían distinguir grados de automatización diferenciados en cuatro categorías, tal como explica Marco Cristoforo, director de Maritime Digital:

- El grado 1 es el que se da hoy en día en la mayoría de los buques de navegación. Corresponde con barcos en los que la tripulación es indispensable únicamente cierta parte de las operaciones está automatizada, sobre todo para aumentar la eficiencia
- El grado 2 seguiría teniendo tripulación, pero la embarcación estaría controlada y operada desde una ubicación remota. Algo fundamental en este nivel es la conectividad a nivel global, la cual debe tener un ancho de banda dedicado para aplicaciones específicas de este estilo que sean de vital importancia.
- En el grado 3 el barco también es controlado desde otro lugar, pero no habría marineros a bordo, lo que haría aún más crucial la necesidad de conexión en todo momento con el centro de mandos, y en el menor tiempo posible con baja latencia, ya que no habría tripulación para tomar el control de la situación y solucionar problemas en casos de emergencia. La tecnología para desarrollar estas condiciones está disponible hoy en día, pero aún no es aplicable en el entorno marítimo.

- Finalmente, el grado 4 sería el alcanzado por el Mayflower, un barco totalmente autónomo que toma todas las decisiones de navegación por sí mismo, y no requeriría la conectividad en todo momento con tierra. Esto hace que la latencia no sea un problema, a menos que el barco se aproxime a puerto, donde sí necesitaría una realimentación rápida de la información, o directamente se maneje mediante control remoto.

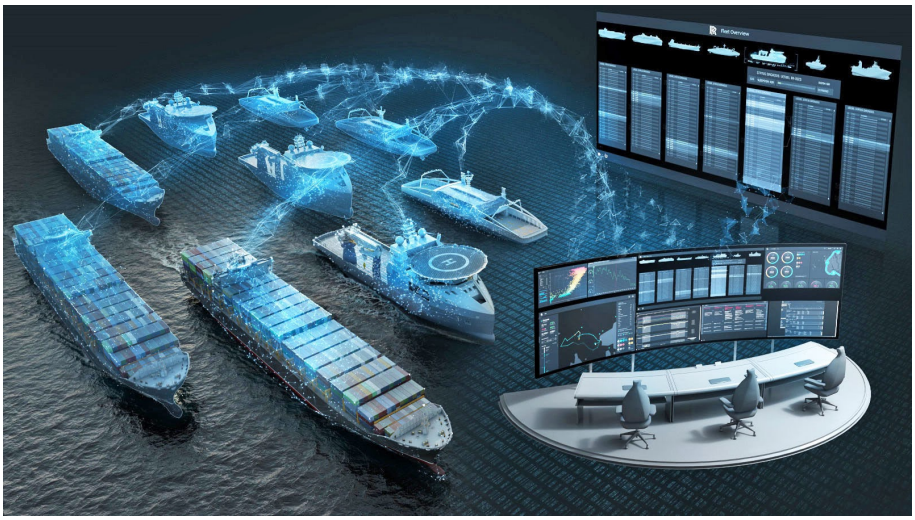


Figura 2.29 Simulación visual del modelo de barcos autónomos propuesto por la inicitiva de financiación europea Autoship[14].

La automatización de cualquier tipo de vehículo ya sea terrestre, marítimo o aéreo es sin duda una de las cuestiones de vanguardia a las que se va a tener que enfrentar la sociedad en un futuro más próximo de lo que parece, donde la motivación principal debe ser el bienestar y la seguridad de las personas que la conforman, y no la reducción de personal y costes. Por ello es necesario prestar atención a todas estas noticias acerca de innovaciones que pueden reflejar avances venideros, ya que reflejarán cambios importantes en las vidas de las personas.

[14]Fuente: *https://sectormaritimo.es/proyecto-autoship-buques-autonomos-que-cambiaran-el-mundo-del-transporte*, [accedido 1 Jul, 2021]

3 Metodología

"If you're a lazy and not-too-bright computer scientist, machine learning is the ideal occupation, because learning algorithms do all the work but let you take all the credit."

— Pedro Domingos

Este capítulo se enfocará en profundizar conocimientos y aclarar conceptos acerca del *Machine Learning* y *Deep Learning* y sus tipos de aprendizaje, así como del diseño y la estructura interna de las redes neuronales, elaborando un proceso escalonado de complejidad en el que se comenzará analizando la neurona, componente principal de las redes a las que da nombre.

Tras ello, se tendrá una base lo suficientemente consolidada para profundizar en la red *YOLO*, fundamental para el desarrollo de este proyecto, así como iniciar y preparar el entorno de trabajo para que todo se logre ejecutar de forma correcta.

3.1 *Machine Learning*

En la década de los 80 surgió una rama de la Inteligencia Artificial conocida como *Machine Learning* (ML) o Aprendizaje Automático, caracterizada por el uso de algoritmos matemáticos que permiten a las máquinas adquirir conocimientos, aprendiendo de datos introducidos para luego aplicarlo y sacar conclusiones de nuevos datos [24]. Pretende hacer de forma automáticas operaciones que en condiciones normales requeriría la intervención humana, suponiendo una gran ventaja a la hora de controlar una cantidad masiva de información o de datos. Lo que es designado como *aprendizaje* corresponde con la capacidad del sistema para identificar una extensa serie de patrones mediante parámetros que los definen. Es decir, la máquina no puede aprender por sí misma, sino que lo hace el algoritmo programado para ello, el cual se modifica y ajusta mediante la entrada de datos en la función en un proceso conocido como entrenamiento. En base a esto adquiere la habilidad para predecir escenarios futuros o tomar decisiones de manera automática sin intervención humana, por ende su nombre [17].

El *Machine Learning* y el *Deep Learning* son conceptos similares, no obstante el primero contempla al segundo ya que se trata de un planteamiento más general. El *Deep Learning* corresponde con aquellos algoritmos concretos de *Machine Learning* que basan su estructura en modelos del cerebro humano, replicando las conexiones y procesos neuronales biológicos [24].

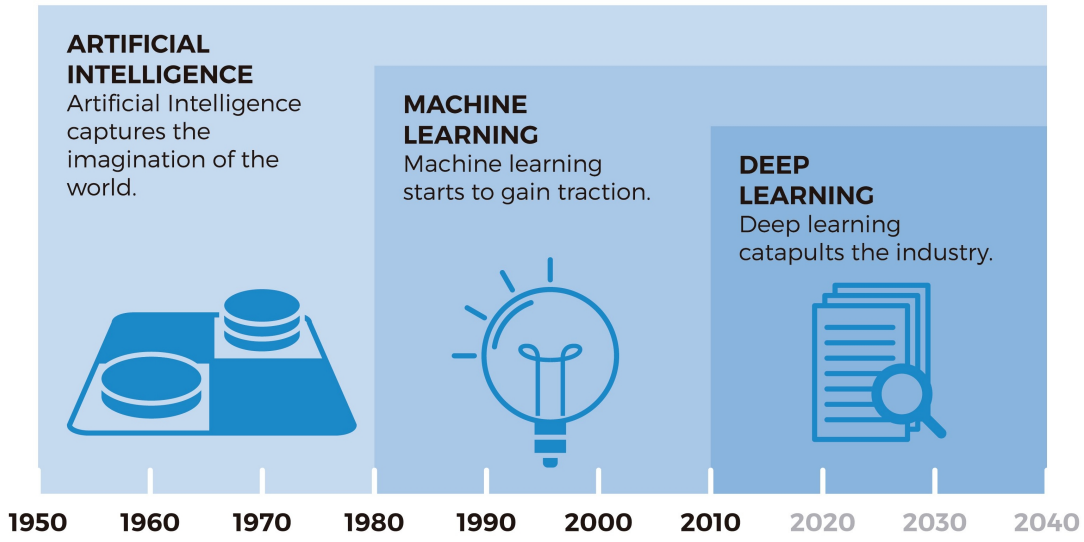


Figura 3.1 Esquema gráfico de los conceptos tratados en el apartado [24].

En consecuencia, a partir de un gran número de ejemplos de las diferentes situaciones a las que se quiere que contribuya la máquina, puede formularse un modelo que generalice un comportamiento previamente observado en las muestras, para posteriormente realizar predicciones sobre casos totalmente nuevos. A pesar de que los algoritmos de aprendizaje automático puedan ser tan variados como coyunturas se dicten, pueden ser categorizados en tres principales tipos de aprendizaje que serán estudiados en el siguiente apartado.

3.2 Algoritmos de entrenamiento

Se ha estudiado que las redes neuronales procesan la información de la entrada para obtener una salida que dé resultados útiles para el problema a resolver. Esto se realiza a través de las funciones que componen la red, utilizando los valores de unos parámetros que deben ser los adecuados para que la salida sea óptima. Para conseguir esto es necesario ajustar los parámetros para que con cada entrada se obtenga la salida deseada, lo cual es posible gracias al proceso de entrenamiento. El entrenamiento es realizado sobre un conjunto de datos de ejemplo para que la red neuronal sea capaz de aprender de ellos, comúnmente conocido como *dataset* de entrenamiento.

Normalmente, durante el entrenamiento, la topología de la red no cambia, sino que lo que sí varían son los pesos de cada una de las conexiones, lo que se conoce como adaptación de los pesos. Se puede establecer que el entrenamiento crea (dándole valores distinto de cero), destruye (dándole valor cero) y modifica las conexiones, como se haría en un sistema biológico real. Debido a que en el transcurso del entrenamiento los valores de los pesos varían de forma notable durante sus cientas de iteraciones, se puede considerar que el entrenamiento ha llegado a su fin cuando los valores de los pesos permanentes relativamente constantes.

Para poder clasificar los mecanismos de aprendizaje es necesario conocer cómo se modifican los pesos hasta conseguir un ajuste correcto, es decir, qué criterios se siguen para determinar el valor del peso de las conexiones para conseguir que la red aprenda nueva información. Todo esto puede verse reflejado en [60]. Normalmente se suelen clasificar en dos grupos: aprendizaje supervisado y aprendizaje no supervisado. Una distinción importante es también si la red es capaz de aprender en su uso habitual, siendo un aprendizaje *online*, o si, por el contrario, los pesos de

las conexiones permanecen fijos durante su funcionamiento, pudiendo variar exclusivamente en su fase de entrenamiento, siendo un aprendizaje *offline*. La ventaja de tener unos pesos estáticos es que su funcionamiento es más estable y constante. Una generalización de las reglas de entrenamiento podría ser la siguiente:

$$Peso\,nuevo = Peso\,antiguo + Cambio\,en\,el\,peso \tag{3.1}$$

Que escrito en formato matemático quedaría de la siguiente forma:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t) \tag{3.2}$$

Siendo t la iteración actual, y w_{ij} un peso concreto que se está ajustando.

3.2.1 Aprendizaje supervisado

Consiste en un método de aprendizaje en el cual se realiza un entrenamiento con un agente externo que determina si el resultado de salida es el correcto ante una determinada entrada. Si la salida no es la deseada, este agente actuará sobre los pesos, modificándolos, con el fin de que en futuras iteraciones la red sea capaz de obtener la salida conveniente. El entrenamiento se realiza proporcionando cierta cantidad de datos que ya contienen las características y clases que se desea que la red aprenda, y están etiquetados al detalle, posibilitando que la red adquiera conocimientos sobre resultados contrastados [17].

Pueden diferenciarse dos sistemas de aprendizaje supervisado. En primer lugar, el sistema de **clasificación** da como resultado una clase entre un número limitado de clases [45], o categorías para las que la red ha sido entrenada. Puede ser implementado en aplicaciones muy diversas como identificación de dígitos, diagnósticos o detección de fraude de identidad entre otros [17]. Por su parte, cuando se usa un sistema de **regresión**, el resultado en la salida es un número, es decir, un valor numérico de todos los casos infinitos que hay. Ejemplos de aplicaciones para este método podrían ser en predicciones meteorológicas o expectativas de ventas, tiempo o crecimiento [45].

Dentro del aprendizaje supervisado se pueden dar distintos tipos de algoritmos para realizar el entrenamiento de la red una vez definida cómo será su salida. Dos de los más importantes son mostrados a continuación:

- **Aprendizaje por corrección de error.**

  En este procedimiento de ajuste se tiene en cuenta el error cometido entre la salida de la red neuronal y el valor esperado, y se realizan correcciones en los pesos de las conexiones para intentar disminuir este error.

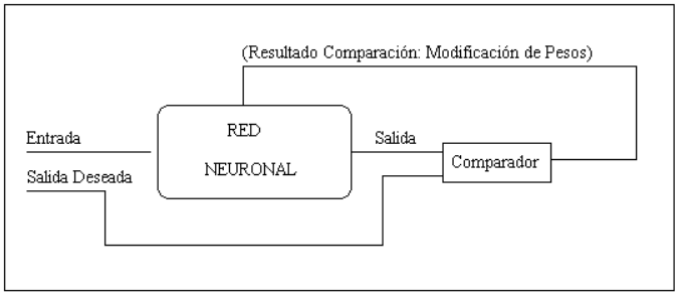


Figura 3.2 Modelo visual del algoritmo de aprendizaje por corrección de error.

Un algoritmo muy importante en este ámbito es el de *backpropagation*. Parte de la idea de aprendizaje *delta*, que es como se conoce al algoritmo *Least Mean Squared (LMS) Error*, el cual calcula la desviación del error en la salida, teniendo en cuenta también las salidas de todas las neuronas anteriores. El procedimiento proporciona una buena información del error que se comete, y con esto, un aprendizaje más rápido. El *backpropagation*, por tanto, contempla una regla LMS multicapa, permitiendo por primera vez ajustar pesos de las conexiones de las capas ocultas.

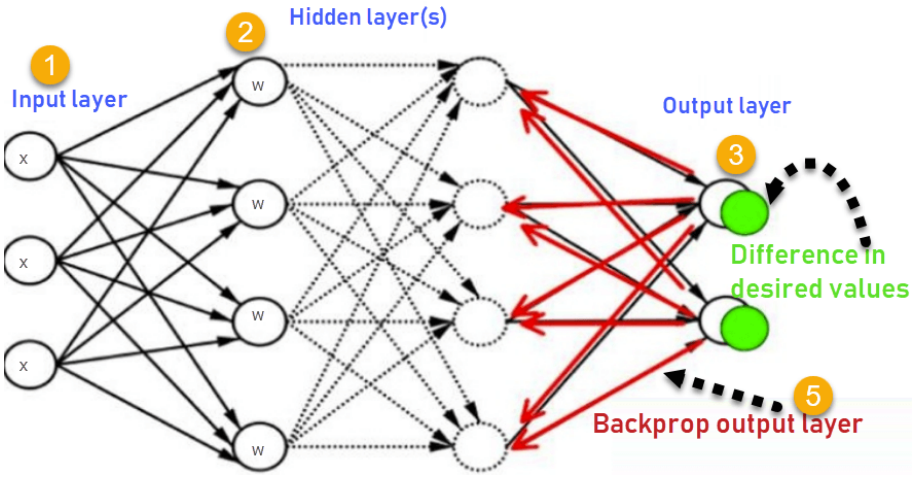


Figura 3.3 Modelo visual del algoritmo de *backpropagation*[1].

- **Aprendizaje estocástico.**

  Como su propio nombre indica, este tipo de aprendizaje está sometido al azar. Se realizan cambios aleatorios en los pesos de las conexiones neuronales, y se analiza el resultado de la salida comparándola con la respuesta objetivo con distribuciones de probabilidad.

3.2.2 Aprendizaje no supervisado

Es un tipo de aprendizaje en el que no existe un agente externo que determine si la predicción ha sido la correcta, por lo que el ajuste de los pesos se realiza de forma automática. No va a recibir ninguna señal del entorno que indique una respuesta objetivo.

Son redes que deben encontrar por sí mismas las similitudes, categorías y características comunes en sus datos de entrada, algo que complica el trabajo ya que no se puede saber si se puede seguir mejorando la red a una versión más correcta.

Una de las principales aplicaciones del algoritmo no supervisado se basa en el agrupamiento o ***clustering***, decidiendo, por ejemplo, el número de clases que ve dadas las similitudes de los bancos de datos. Un caso aplicado a la vida real sería para las compañías la agrupación de clientes que se comporten de una determinada manera, buscando subgrupos dentro de los datos y creando un número de categorías, en principio, desconocido. El dato llamativo de este tipo de algoritmos es que la red será capaz de buscar similitudes entre los datos, pero no se sabe por medio de qué característica los agrupará, por lo que en lugar de buscar una respuesta objetivo en la salida lo que se pretende es obtener información desconocida del conjunto de datos.

[1] Fuente: *https://www.guru99.com/images/1/030819_0937_BackPropaga1.png*

3.2.3 Aprendizaje por refuerzo

Vistos los dos tipos de aprendizajes anteriores, cabe destacar un tercer tipo que puede ser considerado como una mezcla de ambos, ya que comparte ciertas similitudes pero también grandes diferencias. En el *Reinforcement Learning* o aprendizaje por refuerzo no se dispone de una expresión de salida, por lo que no puede ser considerado aprendizaje supervisado, ya que además aprenden de forma autónoma. Sin embargo, al no consistir en clasificar grupos dadas una muestras de entrada tampoco sería plenamente apropiado ser considerado como aprendizaje no supervisado, quedando, por tanto, en una combinación de ambos [1].

Figura 3.4 Esquema comparativo de *Reinforcement Learning* con los otros dos métodos de aprendizaje explicados[2].

En los casos anteriores se ha constatado cómo los escenarios son algo específicos; por ejemplo se podría detectar si un semáforo está en verde o en rojo, pero para conducir un coche autónomo sería necesario tener en cuenta muchísimos más factores: velocidad, ruta, destino, condiciones lumínicas, etc. Una opción para solucionar esto serían varias redes supervisadas que trabajasen en consonancia, a lo que, en contraste, el *Reinforcement Learning* propone un cambio de enfoque, dotando a la máquina de libertad para tomar decisiones, y haciéndole aprender con una serie de premios o castigos, hasta alcanzar una mejora en sus acciones.

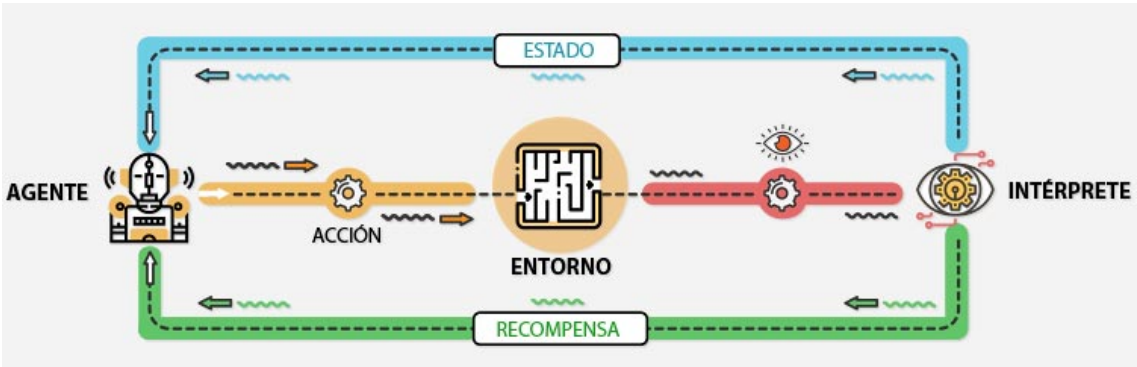


Figura 3.5 Esquema completo de *Reinforcement Learning*[3].

Un cambio importante es que la red aprenderá del entorno en el que se ejecute, y no solamente durante su entrenamiento, lo que antes llamamos como aprendizaje *online*. Hay, por tanto, un nuevo

[2] Fuente: *https://i1.wp.com/www.aprendemachinelearning.com/wp-content/uploads/2020/12/areas-ml.png?w=940&ssl=1*

[3] Fuente: *https://www.iic.uam.es/wp-content/uploads/2018/12/aprendizaje-refuerzo.jpg*

componente, ya que, además del agente externo que ayuda en el entrenamiento en base de premios o castigos en cada acción que tome la máquina, también se dispone del entorno, que es donde el agente actuará, y donde se encontrarán todas las limitaciones, estados y situaciones que la red debe aprender. Esto puede verse reflejado en el esquema de la Figura 3.5.

3.3 La neurona

Las redes neuronales están compuestas, como su propio nombre indica, de gran cantidad de neuronas artificiales, que son las encargadas de llevar a cabo los cálculos e implementar las funciones que caracterizan a la red para dar una salida apropiada a las exigencias demandadas.

El caso más simple de red neuronal es el perceptrón de una capa, el cual consta únicamente de una neurona. Fue propuesto por Frank Rosenblatt en 1957 para ajustar los parámetros libres en un procedimiento de aprendizaje. En el perceptrón, una función lineal es capaz de combinar unos pesos almacenados con los datos de las entradas, siendo posible realizar una clasificación binaria, es decir, de sólo dos clases. La salida binaria se obtiene al aplicar la función de activación de escalón unitario, siendo posible la clasificación de más de dos clases al expandir la salida del perceptrón con más neuronas. Pero para que el resultado sea el deseado las clases deben ser linealmente separables, es decir, estar a ambos lado de un hiperplano [30].

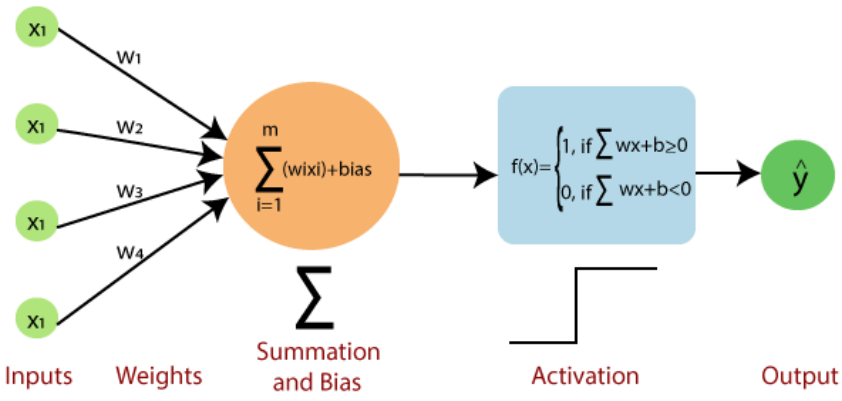


Figura 3.6 Representación gráfica de un perceptrón de una sola neurona[4].

Las neuronas constan de los siguientes elementos [31]:

- Variables de entrada: debe constar al menos de una, y suelen ser representadas vectorialmente como $x = [x_1, x_2, x_3, ..., x_n]$, siendo n el número de entradas. En una neurona biológica haría referencia a las dendritas [8].
- Pesos sinápticos: a cada variable de entrada se le asigna un peso, es decir, un indicador de la importancia que tiene esa conexión. Este peso corresponde a un número, el cual va evolucionando con el entrenamiento, y almacenando así valores que hagan que la red sirva para un propósito u otro. Suelen ser representados vectorialmente como $w = [w_1, w_2, w_3, ..., w_n]$.
- Función de propagación: consiste en la operación que se realiza con las entradas y los pesos sinápticos. En los perceptrones esta función, como ya se ha comentado, es lineal, y consiste

[4] Fuente: *https://miro.medium.com/max/799/1*_Epn1FopggsgvwgyDA4o8w.png*

en la suma de la multiplicación de los pesos por las entradas, suma ponderada, siguiendo la siguiente forma matemática:

$$y = \sum_{i=1}^{n} x_i w_i + b \tag{3.3}$$

Donde todos las variables ya han sido mencionadas, menos b, la cual corresponde al sesgo, siendo este una manera de impulsar la predicción para algunos valores de las entradas. Por ejemplo, si queremos forzar valores de predicción para ciertas entradas que de otra manera no dieran el resultado deseado [6].

La función de propagación no tiene por qué ser lineal, pero suelen serlo en la mayoría de los casos. La regla de propagación más usada habitualmente es la distancia euclídea, usada en redes como Mapas de características Auto-Organizativos (SOM) o las Funciones de Base Radial (RBF).

- Función de activación: la salida de la función de propagación es filtrada a través de la función de activación, que es la que determinará si se activa o no la salida de la neurona.

  La función de activación varía según la capa en la que se encuentre la neurona, dependiendo también de para qué se desea entrenar la red. Las funciones más usadas actualmente para este cometido son la tangente hiperbólica, la función sigmoide, la función escalón y la ReLU (unidad lineal rectificadora).

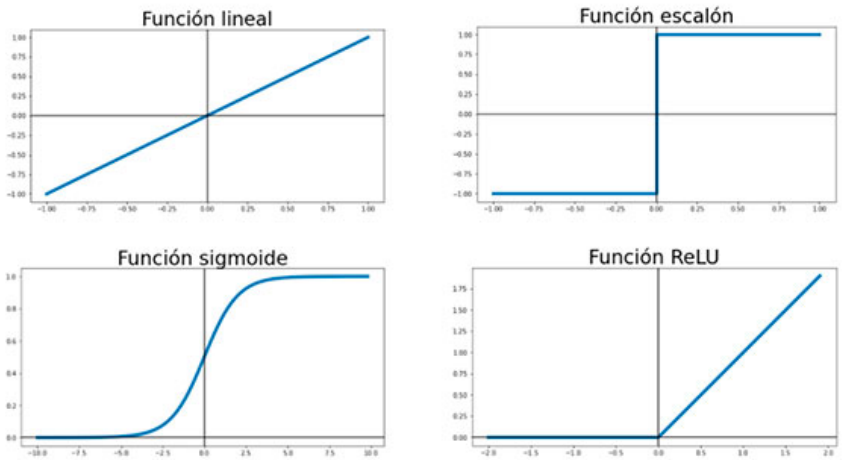


Figura 3.7 Ejemplos de funciones de activación en redes neuronales[5].

Las funciones de activación son la manera de transmitir la información aportada por las neuronas a algo entendible y de utilidad para nuestros propósitos. A veces puede que se quiera transmitir esa información sin modificar, por lo que podría usarse la función identidad.

Son funciones que, por lo general, se suelen usar para dar una «no linealidad» al modelo y que pueda ser capaz de resolver problemas más complejos. Si sólo se usaran funciones de activación lineales, la red neuronal resultante sería análoga a una red neuronal sin capas ocultas, que veremos a continuación [12].

[5] Fuente: *https://www.futurespace.es/wp-content/uploads/2021/04/Funciones-de-activacion.jpg*

3.4 Estructuras de redes neuronales

El perceptrón de una sola capa no es capaz de resolver problemas complejos con grandes datos, o problemas no lineales, por lo que los investigadores desarrollaron redes neuronales en las que se interconectaran un elevado número de neuronas, partiendo de la idea del perceptrón simple [13].

El vector de entrada en lugar de pasar únicamente por una neurona, lo haría a través de cierto número de ellas, realizando las operaciones pertinentes en cada una, y obteniendo un vector de salida más trabajado.

Es a esto lo que se conoce como redes neuronales multicapa, ya que la entrada debe atravesar más de una capa de neuronas, generalmente emplazadas en la salida de la anterior. Las ramificaciones de salida de ciertas neuronas son las entradas de las siguientes, siguiendo así una sucesión por toda la red [12].

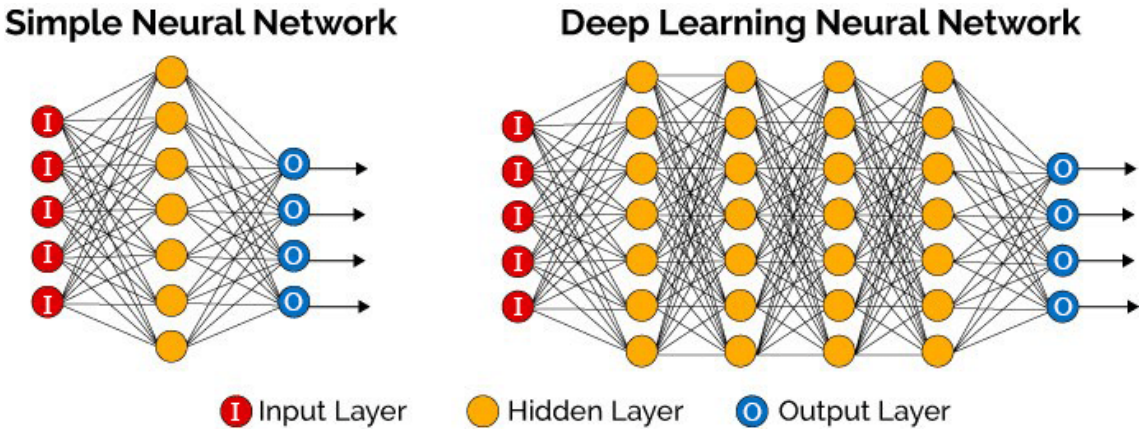


Figura 3.8 Modelo de red neuronal simple y profunda[6].

En la Figura 3.8 se puede observar cómo la diferencia principal entre las redes neuronales simples y las profundas es la cantidad de capas ocultas (neuronas de color amarillo en la imagen). Observamos que las neuronas de color rojo pertenecen a la capa de entrada, las cuales reciben la información o estímulos iniciales del exterior, por lo que no tienen neuronas que las precedan. En cambio, las neuronas de color azul corresponden a las de la capa de salida, por lo que su salida no está conectada a ningunas otras neuronas, ya que consiste en el «estímulo final» de la red neuronal.

Los valores de salida de la red neuronal pueden ser de muchos tipos: continuos, binarios, categorías, etc. Dependiendo de la finalidad de la red y del uso que se le vaya a dar es algo que es necesario tener en cuenta a la hora de diseñarla. Es algo que depende también de las funciones de activación, por ejemplo, si se requiere una salida binaria será necesario el uso de una función escalón como activación, o si la salida debe ser positiva puede usarse una función rectificadora. Hay que tener en cuenta también la potencia de cálculo del equipo en el que se vaya a ejecutar la red, ya que funciones como la sigmoidal o la tangente hiperbólica requieren un coste computacional más elevado.

A modo de resumen, tenemos las piezas necesarias para crear una red neuronal completa: la capa de neuronas de entrada, que recoge la información o estímulos del exterior, las capas de neuronas ocultas, que se encargará de transformar esa información usando los pesos almacenados en cada

[6] Fuente: *https://www.researchgate.net/profile/Peter-Shaw-23/publication/335989001/figure/fig1/AS:806424787566592@1569278060165/Shows-a-comparison-of-a-simple-and-deep-neural-network-highlighting-the-different-number.ppm*

neurona, y la capa final, que se encargará de proporcionar una salida específica tras recibir los procesamientos de la capa oculta. En la capa oculta y en la capa de salida se usarán las funciones de activación que mejor se ajusten y mejor resultado den a la resolución del problema planteado. Las redes multicapa son especialmente útiles en operaciones de reconocimiento y en clasificación de patrones.

3.4.1 Redes unidireccionales o *feedforward*

Las redes *feedforward* son las que hemos estado descubriendo y ejemplificando con la Figura 3.8. En este tipo de red cada neurona tiene un nexo directo con cada una de las neuronas de la capa siguiente, entendiendo como siguiente la que se encuentra cada vez más próxima a la capa de salida.

En la capa de entrada, el número de neuronas debe ser igual al número de variables de la colección de datos, mientras que en la capa de salida el número de neuronas será el necesario para identificar la salida de forma correcta. Por ejemplo, si hablamos de un clasificador, la capa de salida tendrá tantas neuronas como clases a identificar se empleen.

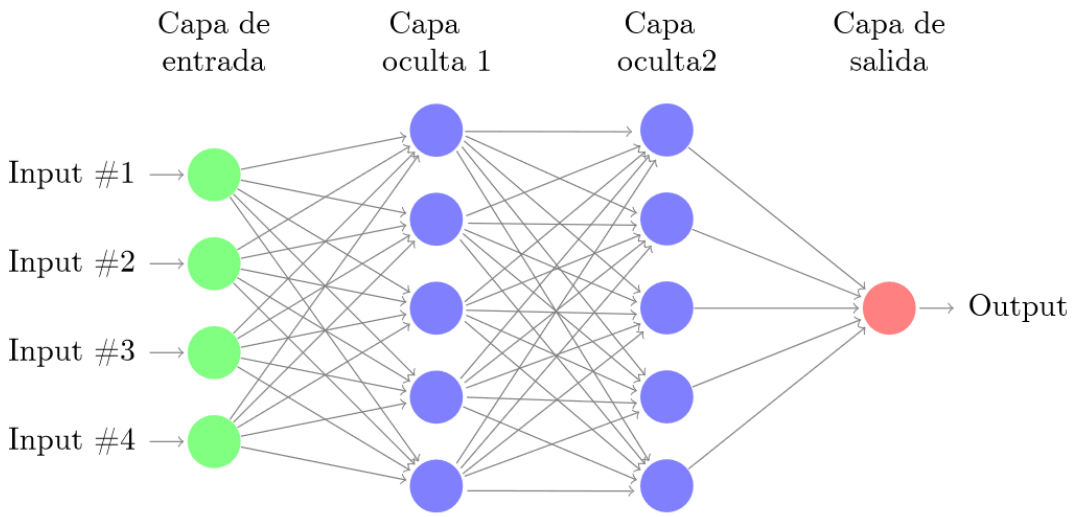


Figura 3.9 Modelo de red neuronal *feedforward* de dos capas ocultas [31].

La salida de una red *feedforward* podría ser calculada identificando los pesos, sesgos y funciones de activación de cada una de las neuronas de las distintas capas. Por ejemplo, si queremos calcular la salida de una capa j, sabemos que que va a depender de su entrada, siendo esta la salida de la capa $j-1$. Si nos vamos al ejemplo de la Figura 3.9, para obtener la salida, *Output*, debemos partir de las entradas, *Inputs*, calculando las salidas de cada una de las neuronas de esta primera capa, a partir de su función de propagación y sus parámetros de pesos y sesgo. A continuación se realizaría el mismo procedimiento con las dos siguientes capas, hasta terminar con la capa de salida que dispone, en este caso, de una única neurona.

A mayor número de capas ocultas, más profunda se considera una red neuronal, cosa que no siempre es deseable, ya que se podría complicar en exceso un problema que no requiera tanta complejidad, obteniendo unos resultados no esperados. Todo esto lo mencionado se puede ver reflejado en [31].

3.4.2 Redes realimentadas o *feedback*

Las redes *feedback* se caracterizan porque la información puede fluir de una capa a otra en cualquier sentido, inluso en el de salida-entrada. Hay algunas que también tienen conexiones laterales entre neuronas de la misma capa, incorporando conexiones estructuradas como excitadores (con pesos positivos), o como inhibidores (con pesos negativos), creando una competencia entre las neuronas de una misma capa [35]. Un ejemplo de ello es el conocido como *winners-takes-all*, en el que a la neurona que da el valor más alto se le asigna el valor total, mientras que a las demás se les asigna el valor de cero [46].

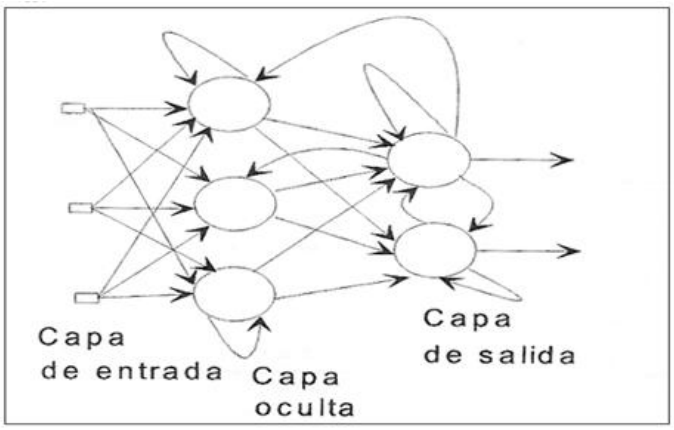


Figura 3.10 Modelo de red neuronal *feedback*[7].

Las redes que pueden propagarse hacia atrás y tienen lazos cerrados, con neuronas que se comunican consigo mismas, son llamadas redes recurrentes [60].

3.4.3 Redes con conexiones residuales

Las redes residuales son aquellas redes que tienen conexiones con capas no consecutivas, es decir, son capaces de "saltarse" capas en la conducción de la información. En el diseño de este tipo de redes se establece conexión entre neuronas de hasta dos o tres capas de diferencia. En la Figura 3.11 podemos apreciar conexiones entre capas no consecutivas, donde la activación de la capa $l-1$ es saltada, pasando directamente desde $l-2$ a l.

Es un tipo de red que surgió en 2015, conocida como ResNet, ganando *challenges* tan prestigiosos como el ImageNet. Su gran innovación reside en que permitió entrenar por primera vez redes muy profundas, de más de 100 capas, sin la problemática conocida como *Vanishing Gradient* durante el ajuste de parámetros, y obteniendo resultados muy buenos [56].

Otro fallo que ayuda a reducir es el problema de *accuracy saturation*, en el cual el aumento de capas en un modelo correcto de modelo profundo produce un aumento en el error de entrenamiento. El hecho de saltarse capas implicaría una red neuronal con menos capas al inicio del entrenamiento, simplificando la red y consiguiendo mejoras en la velocidad, habiendo menos capas por las que se propagaría el error. A medida que se continúa el entrenamiento, la red va restaurando gradualmente las capas pasadas por alto [27].

[7] Fuente: *https://grupo.us.es/gtocoma/pid/pid10/RedesNeuronales_archivos/image117.jpg*
[8] Fuente: *https://upload.wikimedia.org/wikipedia/commons/thumb/5/5f/ResNets.svg/800px-ResNets.svg.png*

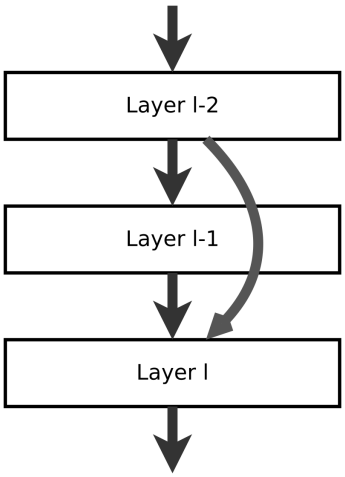


Figura 3.11 Ejemplo conexión residual entre capas[8].

Se usan este tipo de redes en gran frecuencia en los ámbitos de clasificación de imágenes, reconocimiento facial y detección de objetos, siendo una revolución en estos ámbitos [22].

3.5 YOLO

YOLO (*You Only Look Once*) establece un algoritmo de una sola red convolucional capaz de predecir simultáneamente *bounding boxes* y las probabilidades de cada clase directamente desde las imágenes completas únicamente con una evaluación, de ahí su característico nombre. YOLO estudia la imagen entera durante el entrenamiento y durante el test, lo que le permite ver más allá de la simple apariencia del objeto a identificar. Comparándolo por ejemplo con la red *Fast R-CNN*, la red de Redmon comete la mitad de errores que *Fast R-CNN* en parches del fondo de la imagen que se confunden con objetos, gracias a que puede ver todo el contexto de la fotografía.

Es un sistema que logra identificar qué objetos hay y dónde se encuentran en la escena mediante tres procesos principales apreciables en la Figura 3.12. En primer lugar realiza un redimensionado de la imagen de entrada hasta un tamaño de 448x448 píxeles. Esta nueva imagen es pasada por la red convolucional, realizando los cálculos pertinentes para que, por último, sean calculados los porcentajes de confianza de cada clase identificada en la imagen [54].

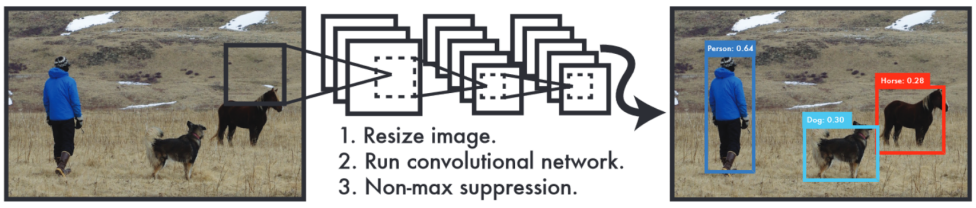


Figura 3.12 Fases del sistema de detección de YOLO [54].

Se trata de un sistema extremadamente rápido, consiguiendo su modelo base procesar imágenes en tiempo real a una velocidad de 45 imágenes por segundo en una GPU Titan X, mientras que una versión más recortada de la red neuronal denominada *Fast Yolo* conseguía la asombrosa velocidad

de 155 *fps* manteniendo incluso una *mean Average Precision* del doble respecto a otros algoritmos de detección en tiempo real. Esto significaría que podría ser capaz de procesar *streamings* de vídeo en tiempo real con una latencia de menos de 25 milisegundos.

Toda la red creada por Redmon es *open source*, por lo que cualquier persona puede aprovechar su código abierto y crear variaciones o mejoras. Algo importante, ya que YOLO también tiene puntos débiles, como puede ser identificar objetos pequeños o su precisión comparada con otros sistemas de detección más centrados en la exactitud, ya que YOLO destaca por su rapidez. Además, sólo es capaz de detectar 49 objetos en una misma imagen, un cifra alta pero mejorable.

El algoritmo divide la imagen de entrada en celdas más pequeñas de dimensión $S \times S$, y si el centro de un objeto a identificar cae dentro de uno de esos cuadros, él será el responsable de la detección de ese objeto. Cada celda podrá predecir B *bounding boxes* y puntuaciones de confianza de cada una de esas cajas, mostrando esta puntuación cómo de segura está la red de que hay un objeto y cómo de cierta cree que es esta predicción.

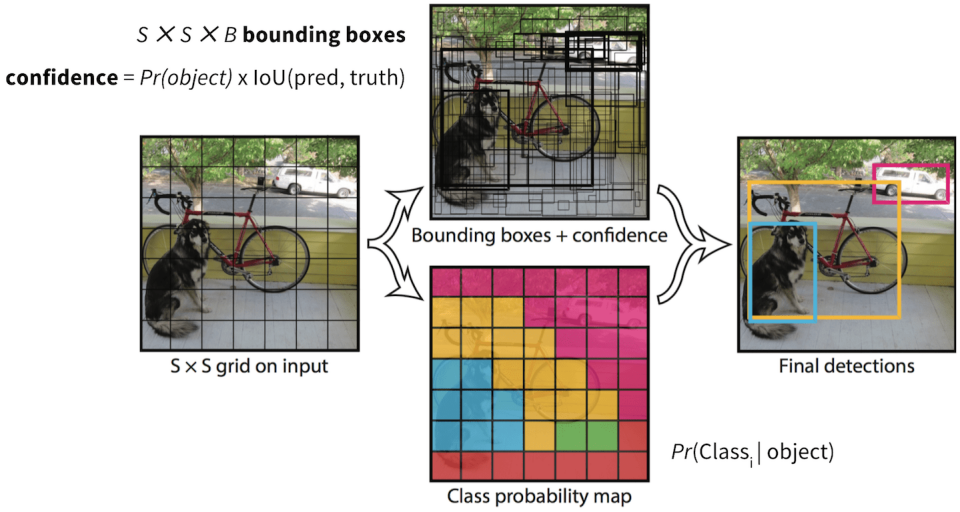


Figura 3.13 Comportamiento de YOLO ante una imagen de entrada [54].

Para obtener la confianza, la red realiza dos cálculos esenciales. En primer lugar se procede a evaluar numéricamente el nivel de fiabilidad que ofrece la red en su predicción, es decir, lo convencida que esté acerca de si un objeto se encuentra dentro del recuadro que se esté examinando. En segundo lugar se procede a realizar la *Intersection over Union* (IoU) de la *bounding box* verdadera sobre la predicha, cuyo procedimiento será explicado a continuación. La multiplicación de ambos términos, como se expone en (3.4), dará como resultado el grado de confianza de la predicción, el cual debe ser cero si no existe objeto alguno en la celda.

$$Obj_{conf} = Pr(Obj) * IoU_{pred}^{truth} \tag{3.4}$$

La *Intersection over Union* es un método usado comúnmente en el ámbito de detección de objetos para cuantificar cómo de preciso está siendo el algoritmo ante un conjunto de datos concretos, así como para calcular su rendimiento. Para poder calcular este estándar es necesario conocer dos cosas de cada imagen: el cuadro que delimite la verdad fundamental, es decir, el *bounding box* correcto de las imágenes de entrenamiento, normalmente realizado manualmente, así como el *bounding box* obtenido como resultado del paso de la imagen por la red [57].

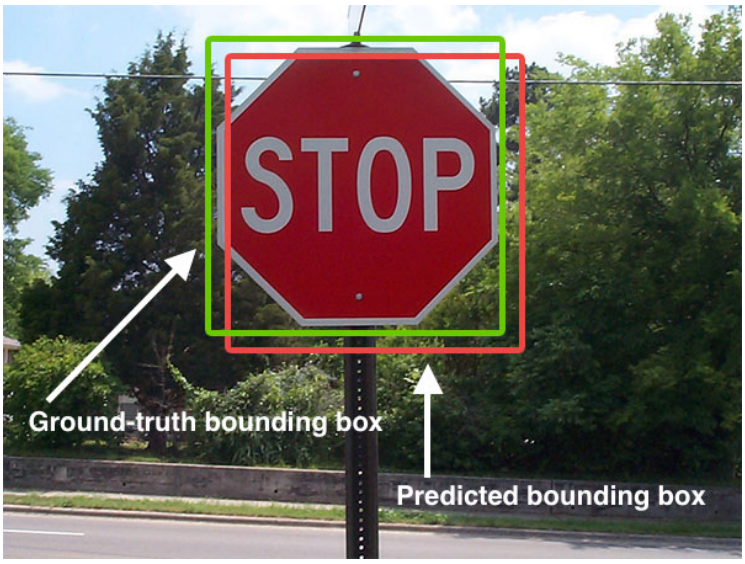


Figura 3.14 Ejemplo de la *bounding box* real (línea verde) frente a la predicha (línea roja) en una señal de tráfico [57].

Teniendo esta información, se puede calcular fácilmente el IoU como una simple división del área que tengan en común ambas *bounding boxes* (área superpuesta), entre el área total del conjunto (área de unión), como se puede apreciar gráficamente con claridad en la Figura 3.15. Por lo tanto, mirando la imagen,se podría deducir fácilmente que el IoU se trata simplemente de un ratio que indica la relación entre ambas áreas de manera numérica.

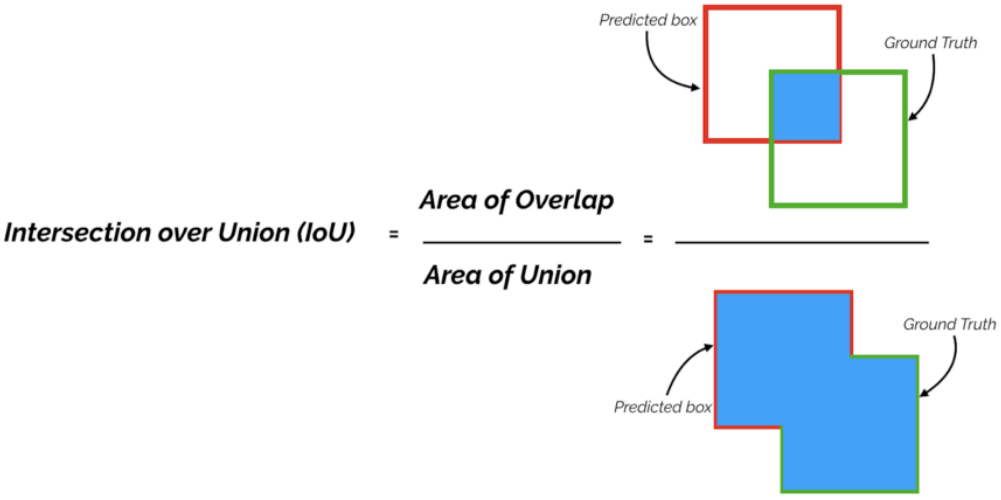


Figura 3.15 Ecuación visual del *Intersection over Union*[9].

El IoU se usa porque la detección de objetos no consiste en una identificación binaria simple, sino en coordenadas de *bounding boxes* predichas que es muy improbable que coincidan exactamente con las verdaderas, por lo tanto es necesaria la búsqueda de una alternativa que determine por cuánto se está equivocando la red.

[9] Fuente: *https://miro.medium.com/max/875/0*-8wov7TFINqw0xAY* [accedido 8 Jun, 2021]

Figura 3.16 Ejemplos numéricos de *Intersection over Union* con distintas *bounding boxes*, real (verde) frente a predicha (rojo) [57].

Además, como se puede comprobar en la Figura 3.16, lo importante no es que la totalidad del rectángulo predicho esté contenido en el real, ya que puede darse casos tales como el primer ejemplo de la imagen, donde la predicción está casi íntegramente contenida en la verdadera, pero hay una área enorme del rectángulo verde que no es cubierta por el rojo, al ser de un tamaño mucho más reducido. Se deduce, por tanto, que lo esencial es la relación de la superficie de la *bounding box* real que es cubierta por la predicha, siendo lógico afirmar, en consecuencia, que las cajas delimitadoras predichas que tengan mayor porcentaje de superposición con la real tendrán mejor puntuación que las que se superpongan menos. Esto hace que este ratio sea excelente para medir la precisión de los algoritmos de detección. Suele ser considerada una buena predicción a partir de un IoU mayor que 0.5, siendo el mejor caso 1, que es cuando se superponen perfectamente [57].

Volviendo al caso particular de YOLO, cada *bounding box* consta de 5 predicciones:

- Coordenadas x, y del centro de la *bounding box* con respecto a la celda.
- Alto h y ancho w de la *bounding box* relativo a la imagen completa.
- La predicción de confianza anteriormente explicada.

Además, por cada celda también se calcula la probabilidad de clase C condicionada a que en la celda haya un objeto, $P(Class_i|Object)$, y, a pesar del número B de *bounding boxes* que pueda predecir cada celda, sólo se calculará una C por celda.

En cuanto al diseño interno de la red neuronal inspirada en GoogLeNet, YOLO cuenta con 24 capas convolucionales, seguida de 2 capas totalmente conectadas. Como módulos de inicio se usan capas de reducción 1×1, con el objetivo de reducir el número de canales de entrada (*input channels*) y con ello el tiempo de entrenamiento, seguidas por capas convolucionales de 3×3. Con esta configuración, las primeras capas convolucionales se encargan de extraer características y rasgos de las imágenes, mientras que las capas totalmente conectadas se encargan de predecir las probabilidades y coordenadas de las *bounding boxes* de salida. La salida final es un tensor de predicciones de $7 \times 7 \times 30$.

La red neuronal fue preentrenada con imágenes de ImageNet, a una resolución de 224×224, usando al inicio únicamente las 20 primeras capas convolucionales de la Figura 3.17 seguidas de una capa de *average-pooling* y una capa totalmente conectada. Tras una semana de entrenamiento, se hicieron una serie de modificaciones y se aumentó el tamaño de las imágenes de entrada para ya centrarse en la detección, pasando a ser de 448×448 para poder obtener más información y detalles de las imágenes, algo importante en la clasificación de objetos.

YOLO usa funciones de activación ReLU en todas las capas, menos en la capa final que usa una función de activación lineal. Toda la información de YOLO para este apartado ha sido sacada de su *paper* oficial [54].

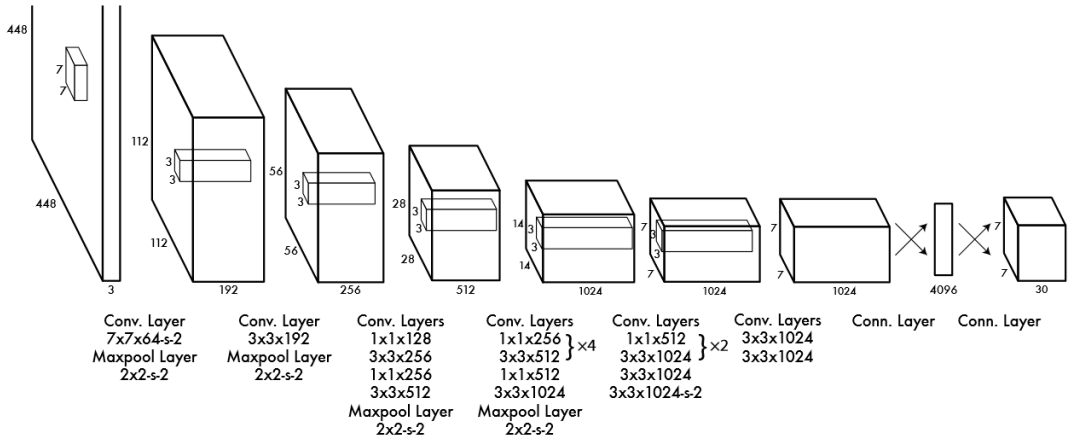


Figura 3.17 Arquitectura a nivel de capas de YOLO [54].

Desde 2016 en que YOLO fue presentada ha ido experimentando gran número de evoluciones y mejoras, sobre todo gracias a que es un proyecto *open source*. Fue en el año 2017 cuando fue presentada la segunda versión de YOLO desarrollada por los programadores de su primera versión, con el nombre de YOLOv2, y prometiendo ser mejor, más rápida y más robusta.

Yolov2 fue entrenada también al inicio con imágenes en tamaño 224×224, pero antes de cambiar a la resolución de 448×448 se hizo un ajuste de entrenamiento de 10 *epochs* también con esta resolución para dar tiempo a la red a trabajar mejor con resoluciones más grandes, lo que desembocó en un incremento de casi un 4 % de mAP.

Esta nueva versión también incorporó los llamados *Anchor Boxes*, que permiten predecir k *bounding boxes* por cada celda en la que se divide la imagen, solucionando el problema de la primera versión en la que cada celda sólo podía detectar un único objeto.

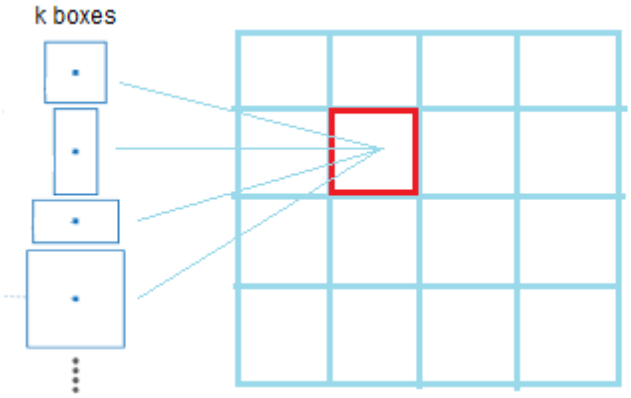


Figura 3.18 Ejemplo visual de la metodología de *Anchor Boxes* en cada celda [40].

Los *Anchor Boxes* son los ratios entre altura y una anchura más repetidos durante el entrenamiento, por lo que cuando la red detecta una *bounding box*, añadirá más *bounding boxes* a su alrededor con las formas que mejor representen al dataset para que sea más fácil para la red neuronal detectar objetos, y siendo $k = 5$ el número de *anchor boxes* en esta versión de YOLO. Vemos por ejemplo en la Figura 3.19 cómo para la celda de la imagen delimitada por la línea roja se proponen cinco *bounding boxes*, recuadros amarillos, con unos tamaños, formas y ratios concretos que son los que puede que mejor se ajusten al objeto, gracias a lo aprendido con la información del *dataset*.

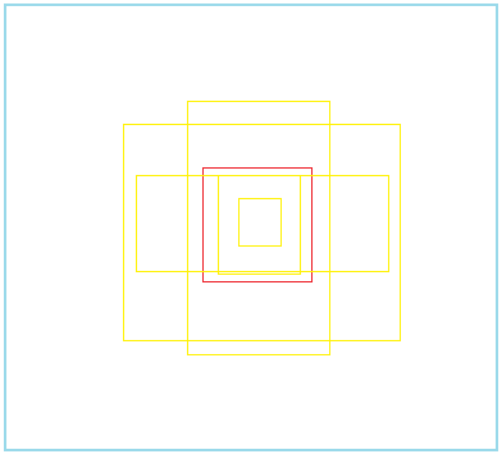


Figura 3.19 Ejemplo de una celda (rojo) con cinco *anchor boxes* (amarillo) [40].

Estas mejoras, entre muchas otras, incrementaron enormemente la precisión de la red, y se consiguió una tasa de *frames* de entre 40-90 *fps*. Como se aprecia en la Figura 3.20, a 40 cuadros por segundos, la precisión de la red mejoraba en torno a un 15 % el resultado obtenido con la primera versión de YOLO, siendo siempre superior en todas las pruebas con esta mejora, incluso a la tasa máxima de *frames*: 90 imágenes por segundo.

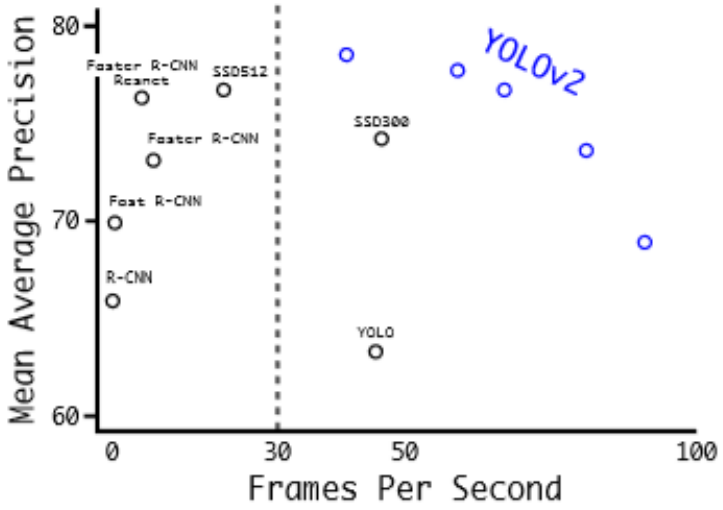


Figura 3.20 Comparación de YOLOv2 frente a otros detectores de objetos, mostrando la precisión medida en mAP frente a tasa de cuadros por segundo [40].

Junto a YOLOv2 también se presentó YOLO9000, el cual está pensado para aquellos casos en los que se necesite detectar más de 20 clases, siendo posible detectar hasta 9000 categorías de objetos, de ahí su nombre, optimizando la detección junto a la clasificación. YOLOv2 fue entrenado para clasificación, y luego fue adaptado para detección, principalmente porque los *dataset* para clasificación y detección son distintos, por lo que con esta versión se propone un método para realizar el entrenamiento tanto de detección como clasificación de forma conjunta. Esto se realiza mezclando durante el entrenamiento imágenes tanto de clasificación como de detección, siendo

el problema más importante que los *datasets* de detección son más escasos y reducidos. Cuando durante el entrenamiento detecta una imagen etiquetada como detección, usa *backpropagation* con toda la red completa de YOLOv2, mientras que cuando recibe una etiqueta de clasificación la *backpropagation* se acota únicamente a las partes de la red diseñadas para clasificación [40].

En 2018 fue presentada la tercera versión de la red, bajo el nombre de YOLOv3. Una de sus novedades destacables es que predice la confianza de cada *bounding box* usando regresión logística, pero sólo acaba asignándole 1 a la que más se sobrepone al recuadro real. Por ejemplo, vemos en la Figura 3.21 dos *bounding boxes* reales en negro, para los que se tienen tres *prior* o *anchor boxes*, en amarillo. Para la *bounding box* real del objeto 1 la *prior box* que más se superpone es la 3, mientras que para la del objeto 2 es la *prior box* 1, por lo que son estas las que se le asigna. Sólo se asigna una *prior box* a cada caja real, y el resto son descartadas. Si la *bounding box* no tiene el IoU más alto pero solapa por más del umbral de 0.5 una *truth box*, se ignora la predicción.

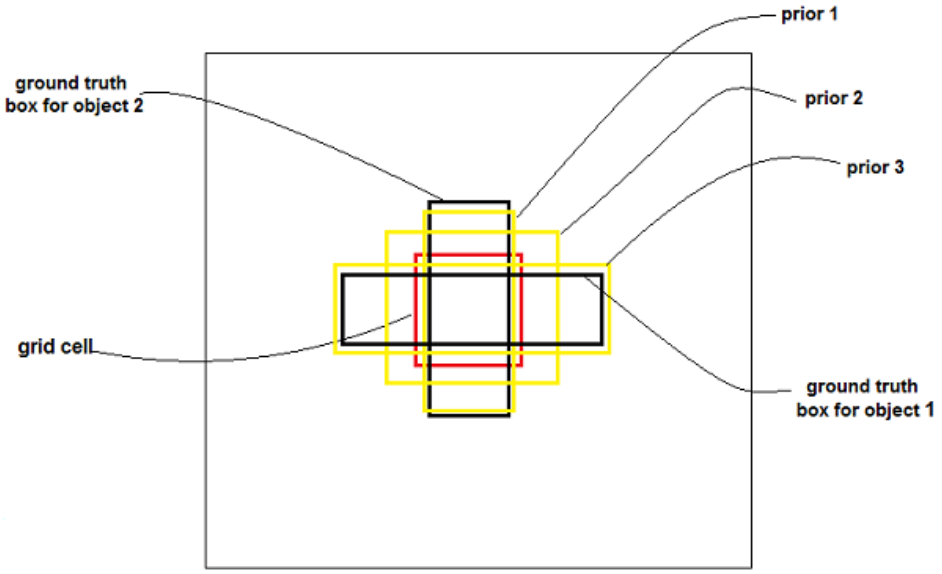


Figura 3.21 Ejemplos de distintas *prior boxes* para dos *truth boxes* de objetos [40].

Una incorporación importante fue el poder predecir etiquetas múltiples. Hay casos en los que una misma *bounding box* puede tener dos clases asociadas, como por ejemplo las etiquetas de persona y mujer o perro y chihuahua. En las versiones anteriores se usaba *softmax* como función de activación, lo que obligaba a que cada recuadro pudiera tener sólo una clase. Es por ello que en YOLOv3 se sustituye por clasificaciones logísticas para cada clase, pudiendo así detectar diferentes etiquetas para un mismo objeto.

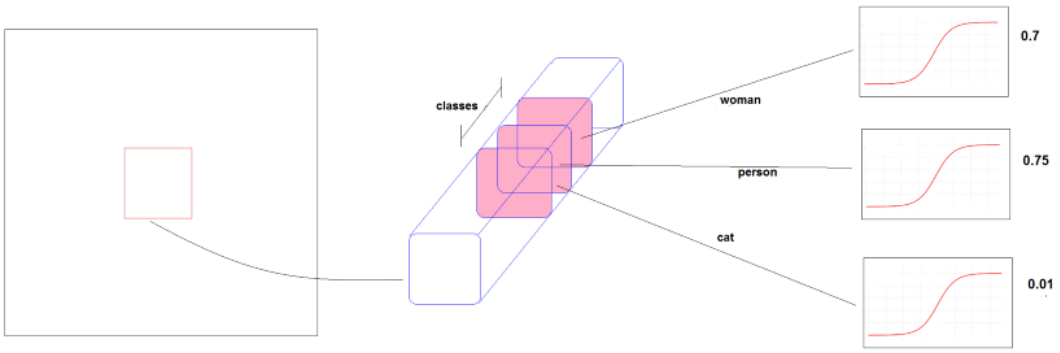


Figura 3.22 Representación visual de etiquetas múltiples para un mismo objeto en YOLOv3 [40].

Otra mejora significativa es su comportamiento con objetos pequeños, el cual supera su desempeño con respecto a las versiones anteriores, gracias a las conocidas como *short cut connections*. Esto permite una mejor obtención de detalles en este tipo de fotos, pero puede afectar a objetos de tamaños medianos y grandes.

Por último, a diferencia de las versiones anteriores de YOLO que predecían la salida en la última capa, este tercer lanzamiento predice *bounding boxes* en tres etapas distintas, apreciable en la Figura 3.23. En cada etapa se proponen tres *anchor boxes* y se predicen tres *bounding boxes* por cada celda, pero cada objeto se sigue asignando a una única celda con un tensor de detección.

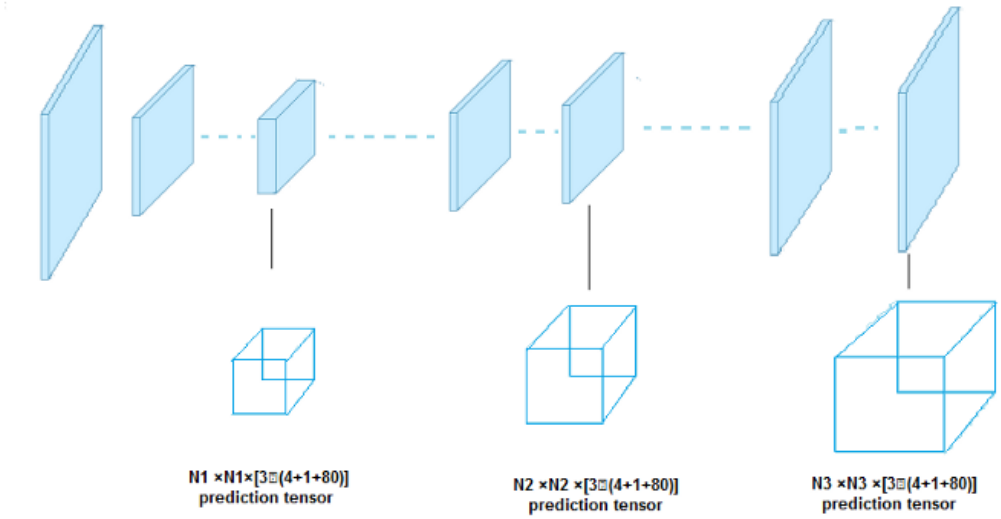


Figura 3.23 Representación visual de las tres etapas en las que se predicen *boxes* con YOLOv3 [40].

YOLO se ha ido convirtiendo en uno de los detectores de objetos más importantes y conocidos en el panorama, sin embargo, en febrero de 2020 su creador, Joseph Redmon, anunció que dejaría de investigar en visión artificial, aludiendo a que estaba viendo impactos negativos en el fruto de su trabajo. Fue a raíz de estas declaraciones cuando la comunidad comenzó a preocuparse sobre el futuro de YOLO, hasta que llegó Alexey Bochkovskiy para coger las riendas del proyecto, y presentando su propio *fork* en abril de 2020 YOLOv4 [62].

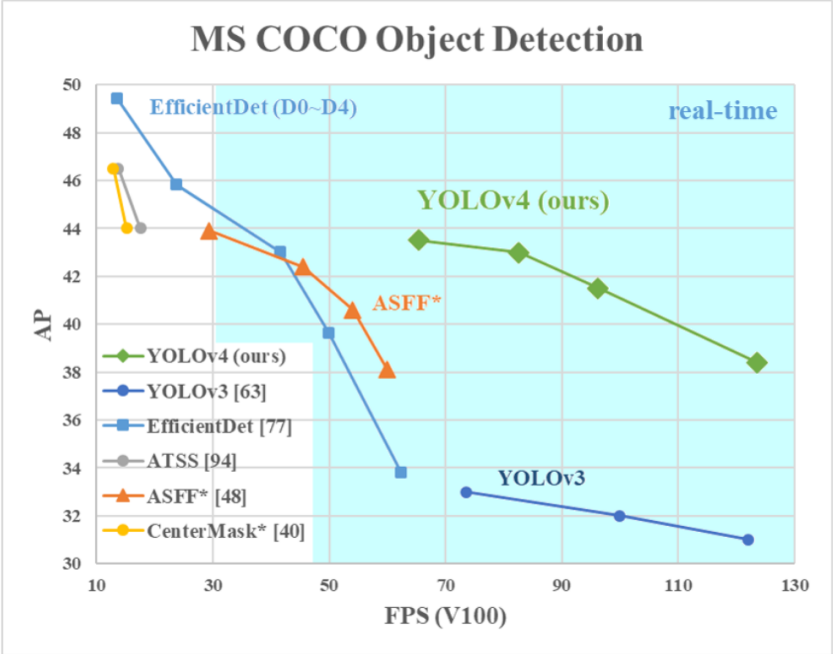


Figura 3.24 Comparación de precisión y velocidad de YOLOv4 con otros detectores de objetos, incluido YOLOv3, usando el dataset de COCO. (Fuente: *Paper* oficial YOLOv4)

Esta nueva versión alcanzó una gran mejora en la precisión y el acierto del reconocimiento comparada con YOLOv3, sin aumentar prácticamente el tiempo de cómputo, pero sí los costes de computación. Consiguió un rendimiento de 65 fps en un Tesla V100, y saliendo victorioso en comparaciones de velocidad y acierto contra otros detectores. Es por ello por lo que esta versión de YOLO será la base para el proyecto que se está desarrollando en este trabajo. A modo de resumen se recalcan en la tabla 3.1 los rendimientos principales de cada versión de YOLO.

Tabla 3.1 Rendimiento de diferentes versiones de YOLO [53], mostrando los FPS al ejecutarse en una GPU Titan X.

Model	Input size	Train set	Test set	mAP	FPS
YOLOv1	448x448	VOC 2007+2012	VOC 2007	63.4 %	45
Fast YOLOv1	448x448	VOC 2007+2012	VOC 2007	52.7 %	155
YOLOv2	416x416	VOC 2007+2012	VOC 2007	76.8 %	67
YOLOv2	544x544	VOC 2007+2012	VOC 2007	78.6 %	40
tiny-YOLOv2	416x416	VOC 2007+2012	VOC 2007	57.1 %	**207**
YOLOv2	608x608	COCO	COCO	48.1 %	40
YOLOv3	320x320	COCO	COCO	51.5 %	45
YOLOv3	416x416	COCO	COCO	55.3 %	35
YOLOv3	608x608	COCO	COCO	57.9 %	20
YOLOv4	608x608	COCO	COCO	65.7 %	23

3.6 Entorno de trabajo

3.6.1 Puesta a punto de la Jetson Nano para visión artificial

Descarga del sistema y *flasheo* de microSD

El primer paso para poder ejecutar YOLO en la Jetson Nano comprendía instalar el sistema operativo Ubuntu 18.04 en su almacenamiento, el cual consistía en una tarjeta de memoria microSD de 32 GB. Para ello era necesaria la descarga de varios elementos, siendo principales la imagen *.iso* del sistema, así como un programa para *flashear* la imagen descargada, en este caso balenaEtcher ya que era el recomendado por la página oficial de Nvidia.

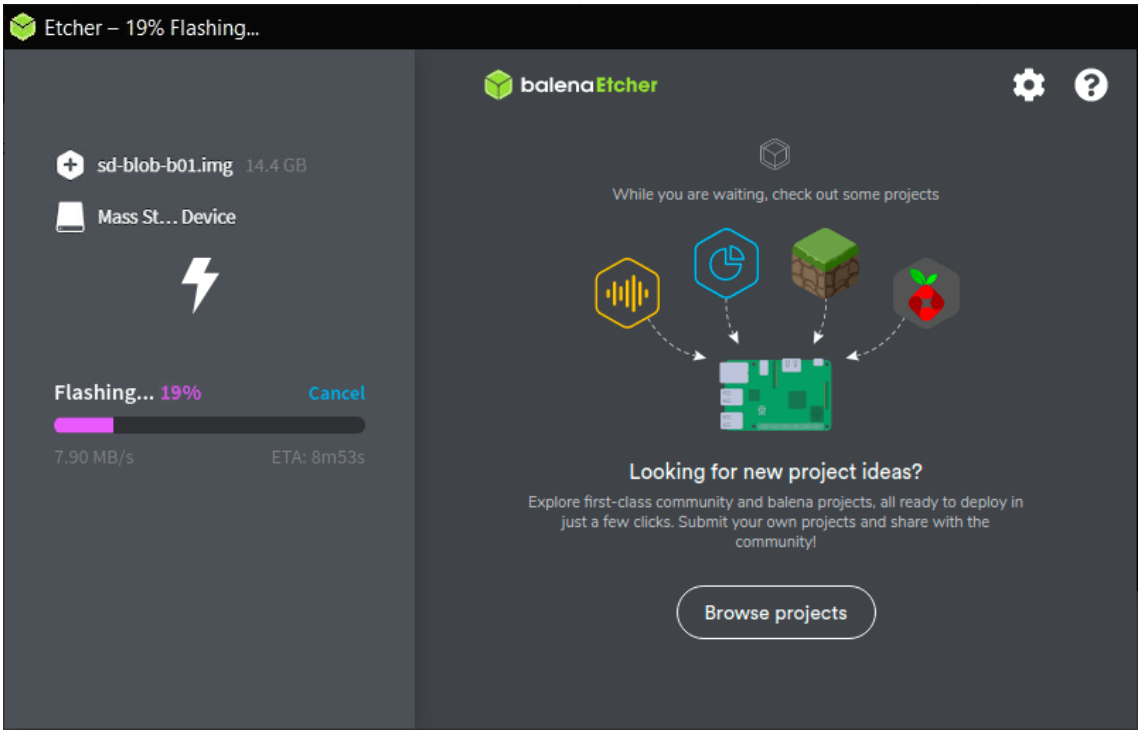


Figura 3.25 Captura de la ventana de balenaEtcher durante el *flasheo* de la tarjeta de memoria.

La imagen del sistema para la Jetson tiene el nombre de JetPack, del cual hay diferentes versiones bajo la misma versión de Ubuntu. A la fecha de este trabajo la última versión disponible era JetPack 4.5.1, pero se ha optado por utilizar una imagen anterior, la JetPack 4.2, para que no haya problemas de compatibilidad entre las versiones de los paquetes y repositorios instalados para su uso en este proyecto de visión por computador.

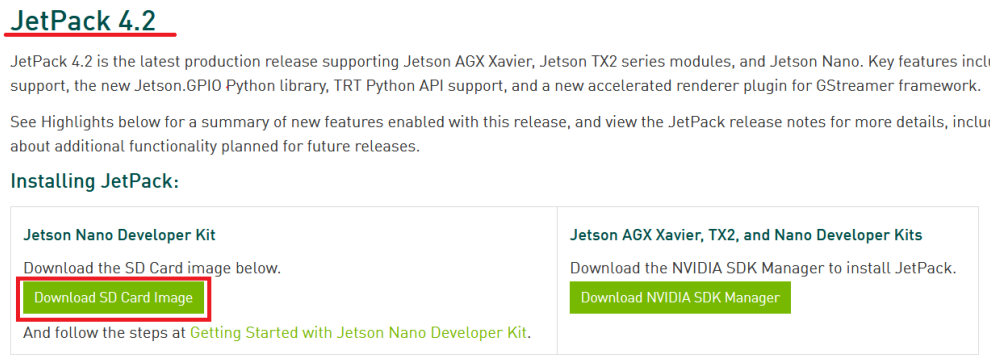


Figura 3.26 Página web de descarga de JetPack 4.2, imagen del sistema operativo para la Jetson Nano.

Una vez *flasheada* la imagen del sistema en la tarjeta de memoria, conectándola al ordenador mediante un adaptador USB, esta debe ser introducida en la ranura de la Jetson, ubicada en la parte inferior de la placa, como se puede ver en la Figura 3.27.

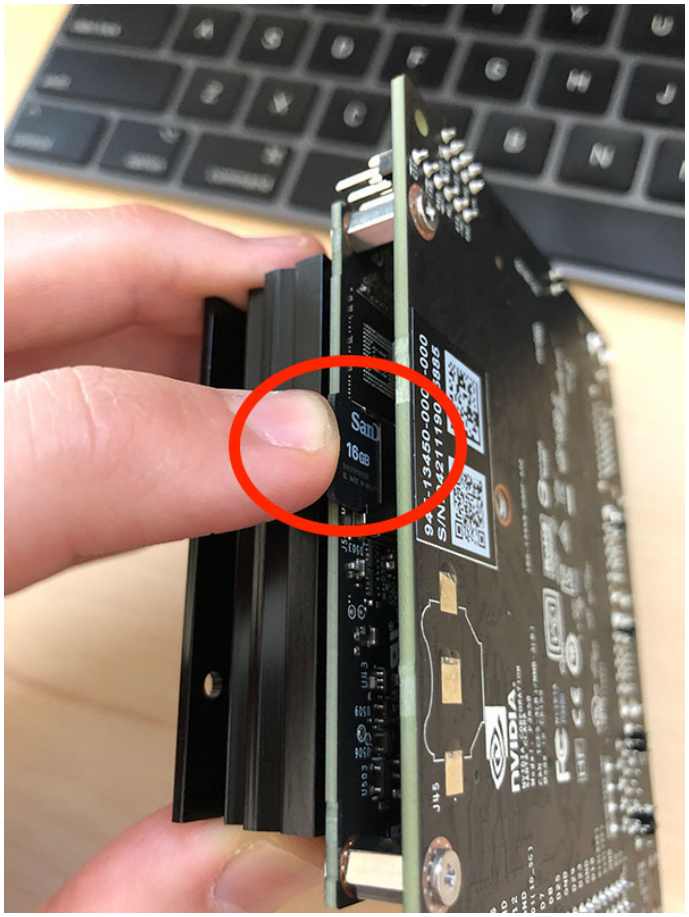


Figura 3.27 Ubicación de la ranura para tarjeta de memoria microSD [58].

Encendido de Jetson Nano y conexión a internet

Después de conectar la placa a la alimentación mediante un conector DC de 12 voltios, se puede transmitir la imagen a una pantalla externa mediante un cable HDMI, obteniendo una visión del escritorio similar a la que se muestra en la Figura 3.28.

Figura 3.28 Imagen del escritorio en el primer encendido del sistema [58].

Tras llegar a este punto, se seguirá el tutorial [59] para completar el largo y tedioso proceso de puesta a punto de la Jetson Nano para su uso en visión por computador y *deep learning*, mediante la instalación de TensorFlow, Keras, TensorRT, y OpenCV entre otros.

Conexión remota

Para facilitar la tarea de instalación de los programas se conectó la placa mediante red local para poder acceder a ella usando SSH o VNC desde un ordenador externo, posibilitando la opción de dejarla instalando programas y paquetes mientras se realizaban otras tareas.

Las instrucciones para activar estas funciones de acceso remoto serían las siguientes:

En primer lugar se debe asegurar de que el servidor VNC está instalado, para ello se ejecutarán los comandos de actualización de repositorios y de instalación del paquete.

```
$ sudo apt update
$ sudo apt install vino
```

Una vez instalado, se debe activar el servidor para que se inicie cada vez que se entra en la sesión, además de activar varios ajustes.

```
$ sudo ln -s ../vino-server.service \
   /usr/lib/systemd/user/graphical-session.target.wants

$ gsettings set org.gnome.Vino prompt-enabled false
$ gsettings set org.gnome.Vino require-encryption false
```

Tras ello, simplemente quedaría introducir los siguientes comandos, en el que `thepassword` indica la contraseña que se desea que tenga para garantizar la seguridad del acceso

```
$ gsettings set org.gnome.Vino authentication-methods "['vnc']"
$ gsettings set org.gnome.Vino vnc-password $(echo -n 'thepassword'|base64)
```

Completado este proceso simplemente quedaría reiniciar el sistema y ya sería accesible desde un escritorio remoto sabiendo la dirección IP local del dispositivo con, por ejemplo, el comando `ifconfig`. Para acceder, como ejemplo, mediante SSH desde la terminal de Ubuntu habría que usar el siguiente comando:

```
$ ssh username@ip_address
```

Donde `username` sería el usuario usado en la sesión iniciada en la Jetson Nano, y la `ip_address` la dirección IP local de la misma. Cuando se introduzca este comando, la terminal pedirá al usuario que ingrese la contraseña establecida en el paso de configuración mencionado anteriormente.

Actualizar sistema y aligerar recursos

El primer paso para aligerar la memoria de la Jetson será eliminar paquetes que requieran gran almacenamiento pero no sean necesarios para la labor que le pretendemos dar, como puede ser el software de ofimática *LibreOffice*. Además, se limpiará el sistema de paquetes obsoletos y se actualizarán los repositorios y los paquetes instalados.

```
$ sudo apt-get purge libreoffice*
$ sudo apt-get clean
$ sudo apt-get update && sudo apt-get upgrade
```

Instalar dependencias básicas

A continuación se realizará la instalación de varias herramientas útiles para desarrollo.

```
$ sudo apt-get install git cmake
$ sudo apt-get install libatlas-base-dev gfortran
$ sudo apt-get install libhdf5-serial-dev hdf5-tools
$ sudo apt-get install python3-dev
$ sudo apt-get install nano locate
```

Así como una serie de prerrequisitos para la instalación de *SciPy*, librerías necesarias para *Cython*, y varias herramientas XML para trabajar con *TensorFlow Object Detection* (TFOD).

```
$ sudo apt-get -yq install libfreetype6-dev python3-setuptools
$ sudo apt-get -yq install protobuf-compiler libprotobuf-dev openssl
$ sudo apt-get -yq install libssl-dev libcurl4-openssl-dev
```

```
$ sudo apt-get -yq install cython3

$ sudo apt-get install libxml2-dev libxslt1-dev
```

Actualizar *CMake*

Para poder compilar de forma correcta OpenCV es necesaria la actualización del compilador *CMake*. Primeramente se descargará desde la página oficial y se descomprimirá el archivador siguiendo las siguientes instrucciones.

```
$ cd ~
$ wget http://www.cmake.org/files/v3.13/cmake-3.13.0.tar.gz
$ tar xpvf cmake-3.13.0.tar.gz cmake-3.13.0/
```

Tras ello, nos iremos al directorio donde se ha descomprimido el archivo y lo compilaremos.

```
$ cd cmake-3.13.0/
$ ./bootstrap --system-curl
$ make -j4
```

Por último, una vez compilado, simplemente queda exportar el directorio al archivo *bashrc* para que se ejecute cada vez que se inicia la terminal. En este caso afecta el nombre de usuario ya que se sitúa en el directorio princial, siendo *Jet* el nombre usado.

```
$ echo 'export PATH=/home/jet/cmake-3.13.0/bin/:$PATH' >> ~/.bashrc
$ source ~/.bashrc
```

Instalar dependencias de desarrollo y de OpenCV

El siguiente paso lógico será la instalación de las dependencias básicas de OpenCV para poder compilar, librerías de imágenes y códecs, así como interfaces gráficas y *Video4Linux* (V4L) para poder conectar *webcams* por USB.

```
$ sudo apt-get -yq install build-essential pkg-config
$ sudo apt-get -yq install libtbb2 libtbb-dev

$ sudo apt-get -yq install libavcodec-dev libavformat-dev libswscale-dev
    libxvidcore-dev libavresample-dev libtiff-dev libjpeg-dev libpng-dev python-
    tk libgtk-3-dev libcanberra-gtk-module libcanberra-gtk3-module libv4l-dev
    libdc1394-22-dev
```

Configurar entornos virtuales de Python

Para tener el proyecto más ordenado e instalado de forma independiente de la instalación principal de *Python* del sistema, se usará un entorno virtual de *Python*, pudiendo observar gráficamente su concepto en la Figura 3.29.

Básicamente permite aislar totalmente diferentes instalaciones de *Python*, pudiendo usar versiones diferentes de él, o de los paquetes que se deseen instalar. Para ello instalaremos en primer lugar la herramienta para gestionar e instalar paquetes en *Python* llamada `pip`, para posteriormente instalar

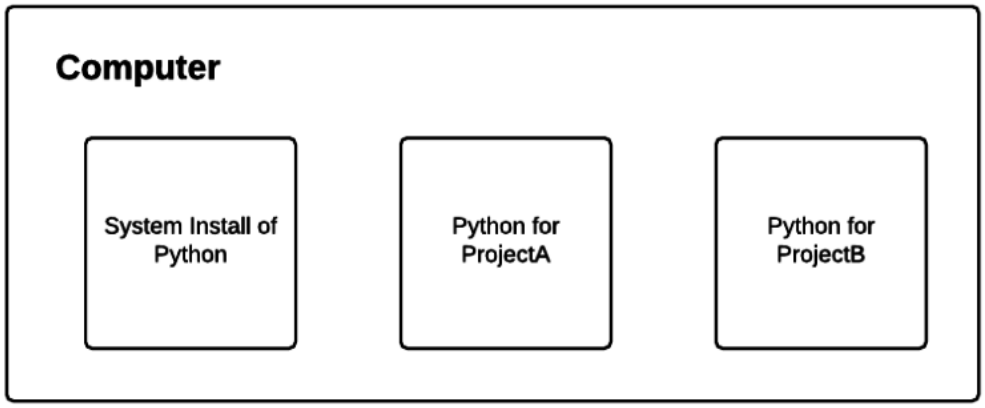


Figura 3.29 Esquema del concepto de entornos virtuales de *Python* [59].

con ella las herramientas `virtualenv` y `virtualenvwrapper` para obtener los entornos virtuales mencionados.

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python3 get-pip.py
$ rm get-pip.py

$ sudo pip install virtualenv virtualenvwrapper
```

Una vez realizado esto, se debe añadir cierta información al archivo *bashrc*, tal como se hizo con *CMake*, para completar totalmente la instalación de *virtualenv*.

```
$ echo '# virtualenv and virtualenvwrapper
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
source /usr/local/bin/virtualenvwrapper.sh' >> ~/.bashrc

$ source ~/.bashrc
```

Creación de un entorno virtual para el proyecto

A partir de ahora, la terminal está preparada para soportar entornos de trabajo diferentes, para los que usaremos los siguientes comandos propios del paquete para trabajar:

- `mkvirtualenv` : Comando para crear un nuevo entorno virtual de *Python*
- `lsvirtualenv` : Comando para listar todos los entornos virtuales creados
- `rmvirtualenv` : Comando para eliminar un entorno virtual creado
- `workon` : Comando para trabajar en un entorno virtual creado, debe ir seguido del nombre del entorno al que se quiera cambiar
- `deactivate` : Comando para salir del entorno virtual actual y cambiar al de por defecto del sistema

Por tanto, se creará un entorno virtual con el comando `mkvirtualenv` con el nombre `py3cv4`, indicando que se usará la versión 3 de *Python* en él, y luego se introducirá el comando para trabajar en él.

```
$ mkvirtualenv py3cv4 -p python3
$ workon py3cv4
```

Todos los siguientes pasos de instalación serán realizados dentro de este entorno virtual para garantizar la independencia con respecto a la instalación original del sistema y aislar los problemas que pudieran surgir.

Instalar el compilador *Protobuf*

El siguiente escalón consistirá en la configuración de *Protocol Buffers*, abreviada como *Protobuf*, la cual se trata de una biblioteca multiplataforma multiplataforma gratuita y de código abierto desarrollada por Google con el objetivo de aumentar la simplicidad y el rendimiento, sirviendo en este proyecto para conseguir que *TensorFlow* sea más rápido. Al ser la Jetson Nano un dispositivo especial de reducidas características, en lugar de realizar la instalación que se realizaría en un ordenador normal mediante `pip` junto con la instalación de *TensorFlow*, se instalará una versión especial adaptada para esta plataforma, tema tratado en el foro de desarrolladores de Nvidia. Por tanto, se descargará en primer lugar una implementación más eficiente del compilador *Protobuf*.

```
$ wget https://raw.githubusercontent.com/jkjung-avt/jetson_nano/master/
    install_protobuf-3.6.1.sh
$ sudo chmod +x install_protobuf-3.6.1.sh
$ ./install_protobuf-3.6.1.sh
```

Este proceso de instalación tomará un tiempo extenso, alrededor de una hora, y una vez realizado hay que instalarlo dentro del entorno virtual, por lo que se realizarán los pasos mostrados a continuación, en el que se usará un archivo `setup.py` en lugar del comando `pip` para compilar el *software* específicamente para la Jetson Nano en lugar de usar un comando generalizado. En resumen, se usa un `setup.py` cuando el rendimiento puede verse afectado y es necesario una adaptación para la Jetson, y se usará `pip` cuando la compilación genérica es más que suficiente y puede correrse sin problemas.

```
$ workon py3cv4
$ cd ~
$ cp -r ~/src/protobuf-3.6.1/python/ .
$ cd python
$ python setup.py install --cpp_implementation
```

Instalar TensorFlow, Keras, NumPy y ScyPy

Tras la instalación de *Protobuf* podemos proceder a la instalación de *Numpy* y *Cython* mediante `pip`.

```
$ pip install numpy cython
```

La versión de *TensorFlow* que se instalará será la 1.13.1 optimizada para la Jetson Nano, ya que, aunque la versión 2.0 está disponible, no es recomendable su instalación debido a que puede haber

una serie de incompatibilidades con la versión de *TensorRT* que viene con el sistema. Además, en esta plataforma de Nvidia tiene problemas de *memory leak* que pueden hacer que la Nano se congele y se cuelgue. Es por ello que, dada la necesidad de un dispositivo estable para el propósito de este proyecto, se instalará la versión 1.13.1. Para ello, se comenzará instalando la versión de *SciPy* 1.3.3, compatible con la versión de *TensorFlow* mencionada. Al instalar una versión de TF adaptada a la Nano, la versión de *SciPy* debe ser también especial, por lo que en lugar de usar `pip` se usará una versión concreta del repositorio en *GitHub*, tal como se recomienda en el *DevTalk de Nvidia*.

```
$ wget https://github.com/scipy/scipy/releases/download/v1.3.3/scipy-1.3.3.tar.gz
$ tar -xzvf scipy-1.3.3.tar.gz scipy-1.3.3
$ cd scipy-1.3.3/
$ python setup.py install
```

Este proceso tardará alrededor de media hora, y una vez realizado se podrá proceder a instalar *TensorFlow* con el uso de la GPU. Un error que surgió durante la instalación por el cual la compilación se abortaba fue que no encontraba el archivo `xlocale.h`, algo que se pudo solucionar mediante un enlace simbólico, y tras ello poder instalar TF con normalidad, así como *Keras* mediante `pip`.

```
$ sudo ln -s /usr/include/locale.h /usr/include/xlocale.h
$ pip install --extra-index-url https://developer.download.nvidia.com/compute/
    redist/jp/v42 tensorflow-gpu==1.13.1+nv19.3
$ pip install keras
```

Instalar la API de *TensorFlow Object Detection*

Una vez finalizada la anterior, el siguiente paso consistirá en instalar la *Application Programming Interface* (API) de *TensorFlow Object Detection* (TFOD), una librería útil para desarrollar modelos de detección de objetos y que servirá para optimizar modelos usando la GPU de la Nano. Para su instalación se clonará el repositorio de *GitHub* de modelos de *TensorFlow*, y luego se cambiará al *commit* que soporta la versión 1.13.1 de TF.

```
cd ~
workon py3cv4
git clone https://github.com/tensorflow/models
cd models && git checkout -q b00783d
```

En este punto se aprovechará también para instalar la API de COCO para trabajar con su *dataset*.

```
cd ~
git clone https://github.com/cocodataset/cocoapi.git
cd cocoapi/PythonAPI
python setup.py install
```

Tras ello, se procederá a compilar las bibliotecas de *Protobuf* usadas por la API de TFOD, las cuales servirán para que las aplicaciones intercambien datos entre ellas de forma simple incluso si están programadas en distintos lenguajes.

```
$ cd ~/models/research/
$ protoc object_detection/protos/*.proto --python_out=.
```

Finalizado este proceso, se creará un archivo *bash* para simplificar el procedimiento necesario cada vez que se quiera usar TFOD API, en la carpeta de usuario y con el nombre *setup.sh*, indicando que es un fichero ejecutable y mostrando al sistema la `PYTHONPATH`.

```
echo '#!/bin/sh
export PYTHONPATH=$PYTHONPATH:/home/`whoami`/models/research:\
/home/`whoami`/models/research/slim' >> ~/setup.sh
```

Instalar los modelos de TF y TRT de Nvidia

Tras la instalación y configuración de lo anterior, se puede proceder a instalar la librería `tf_trt_models` desde su repositorio de *GitHub*, el cual contiene modelos optimizados de *TensorRT* para la Jetson Nano.

```
$ workon py3cv4
$ cd ~
$ git clone --recursive https://github.com/NVIDIA-Jetson/tf_trt_models.git
$ cd tf_trt_models
$ ./install.sh
```

Instalar OpenCV

La última instalación importante será OpenCV 4.1.2 con soporte para CUDA, un conjunto de librerías de Nvidia para trabajar con sus GPUs, ya que correrlo con la GPU en lugar de la CPU hará que los resultados sean más rápidos. *Open Computer Vision* (OpenCV) es una biblioteca libre optimizada para visión por computador en tiempo real, *machine learning* y procesamiento de imágenes, por lo que será importante para el desarrollo del proyecto.

```
$ cd ~
$ wget -O opencv.zip https://github.com/opencv/opencv/archive/4.1.2.zip
$ wget -O opencv_contrib.zip https://github.com/opencv/opencv_contrib/archive
    /4.1.2.zip
$ unzip opencv.zip
$ unzip opencv_contrib.zip
$ mv opencv-4.1.2 opencv
$ mv opencv_contrib-4.1.2 opencv_contrib
```

Una vez descargado del repositorio, descomprimido y renombrado, entraremos al directorio y crearemos una carpeta `build`.

```
$ cd opencv
$ mkdir build
$ cd build
```

Asegurándonos de que estamos en el entorno virtual creado para el proyecto `py3cv4` y en la carpeta `build` creada, usaremos *CMake* para compilar con una serie de *flags*, entre ellas se pueden destacar el uso de `WITH_CUDA=ON`, indicando que usaremos las optimizaciones de *CUDA*, y precisando también con `OPENCV_EXTRA_MODULES_PATH` el directorio de la descarga de `opencv_contrib` renombrado anteriormente.

```
cmake -D CMAKE_BUILD_TYPE=RELEASE \
  -D WITH_CUDA=ON \
  -D CUDA_ARCH_PTX="" \
  -D CUDA_ARCH_BIN="5.3,6.2,7.2" \
  -D WITH_CUBLAS=ON \
  -D WITH_LIBV4L=ON \
  -D BUILD_opencv_python3=ON \
  -D BUILD_opencv_python2=OFF \
  -D BUILD_opencv_java=OFF \
  -D WITH_GSTREAMER=ON \
  -D WITH_GTK=ON \
  -D BUILD_TESTS=OFF \
  -D BUILD_PERF_TESTS=OFF \
  -D BUILD_EXAMPLES=OFF \
  -D OPENCV_ENABLE_NONFREE=ON \
  -D OPENCV_EXTRA_MODULES_PATH=/home/‘whoami‘/opencv_contrib/modules ..
```

Cuando finalice la preparación de la compilación, se obtendrá una salida por la terminal similar a la de la figura 3.30, en la cual se debe inspeccionar cada apartado para asegurar que todo el proceso se ha realizado de forma correcta y sin errores.

```
--     Custom HAL:                  YES (carotene (ver 0.0.1))
--     Protobuf:                    build (3.5.1)
--
--   NVIDIA CUDA:                   YES (ver 10.0, CUFFT CUBLAS)
--     NVIDIA GPU arch:             53 62 72
--     NVIDIA PTX archs:
--
--   cuDNN:                         YES (ver 7.5.0)
--
--   OpenCL:                        YES (no extra features)
--     Include path:                /home/pyimagesearch/opencv/3rdparty/include/
opencl/1.2
--     Link libraries:              Dynamic load
--
--   Python 3:
--     Interpreter:                 /home/pyimagesearch/.virtualenvs/py3cv4/bin/
python3 (ver 3.6.9)
--     Libraries:                   /usr/lib/aarch64-linux-gnu/libpython3.6m.so
(ver 3.6.9)
--     numpy:                       /home/pyimagesearch/.virtualenvs/py3cv4/lib/
python3.6/site-packages/numpy/core/include (ver 1.18.1)
--     install path:                lib/python3.6/site-packages/cv2/python-3.6
--
--   Python (for build):            /usr/bin/python2.7
--
--   Java:
--     ant:                         NO
--     JNI:                         NO
--     Java wrappers:               NO
--     Java tests:                  NO
--
--   Install to:                    /usr/local
-- -----------------------------------------------------------------
--
-- Configuring done
-- Generating done
-- Build files have been written to: /home/pyimagesearch/opencv/build
```

Figura 3.30 Salida de la terminal tras el primer proceso con *CMake* [59].

Una vez comprobado que todo ha salido correctamente y se esté conforme con el resultado, se procederá a terminar la compilación con *Make*:

```
$ make -j4
```

Este proceso final de compilación de OpenCV puede tardar alrededor de 2 horas, terminando con un 100 % y pudiendo ver en la terminal una salida similar a la siguiente.

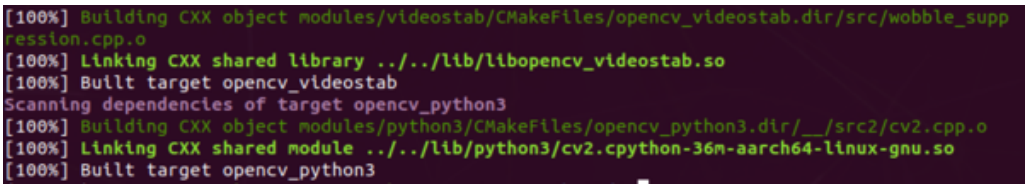


Figura 3.31 Salida de la terminal tras finalizar la compilación de OpenCV [59].

Tras esta larga espera simplemente quedaría instalar lo compilado, y crear un enlace simbólico entre el directorio de instalación de OpenCV y el entorno virtual del proyecto.

```
$ sudo make install

$ cd ~/.virtualenvs/py3cv4/lib/python3.6/site-packages/
$ rm cv2.so
$ ln -s /usr/local/lib/python3.6/site-packages/cv2/python-3.6/cv2.cpython-36m-
    aarch64-linux-gnu.so cv2.so
```

Completado esto, se tendrá finalmente instalado OpenCV de forma oficial, así como la Jetson Nano operativa para su uso en visión artificial.

Instalar otras librerías útiles mediante *pip*

Simplemente se instalarán algunos paquetes extra útiles de *machine learning*, *plotting*, procesamiento de imágenes, y cuadernos de Python entre otros, dentro siempre del entorno virtual `py3cv4`.

```
$ workon py3cv4

$ pip install matplotlib scikit-learn

$ pip install pillow imutils scikit-image

$ pip install dlib

$ pip install flask jupyter

$ pip install lxml progressbar2
```

Probar instalación

Para comprobar que todo ha salido correctamente es conveniente probar todos los componentes instalados, por lo que se comenzará probando la API de TFOD, y tal como se observa en la captura de la terminal en la Figura 3.32 el resultado es positivo.

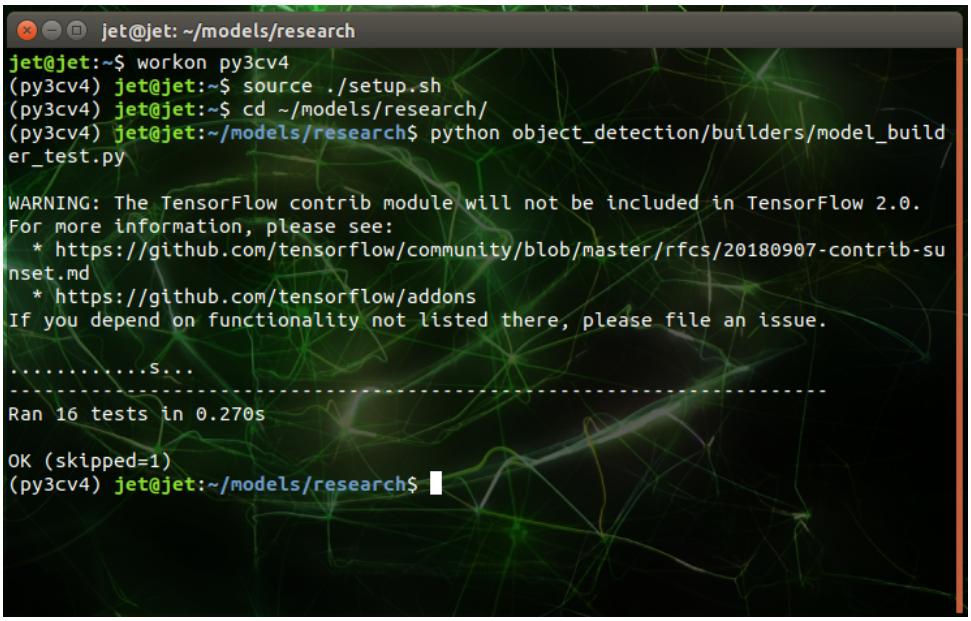


Figura 3.32 Captura de terminal del *test* a la API de TFOD

Lo siguiente que se comprobará será la instalación de OpenCV, para ello se hará uso de una imagen de pingüinos descargada a la que se le insertará un texto, como se ve en la Figura 3.33, para mostrar que todo funciona correctamente.

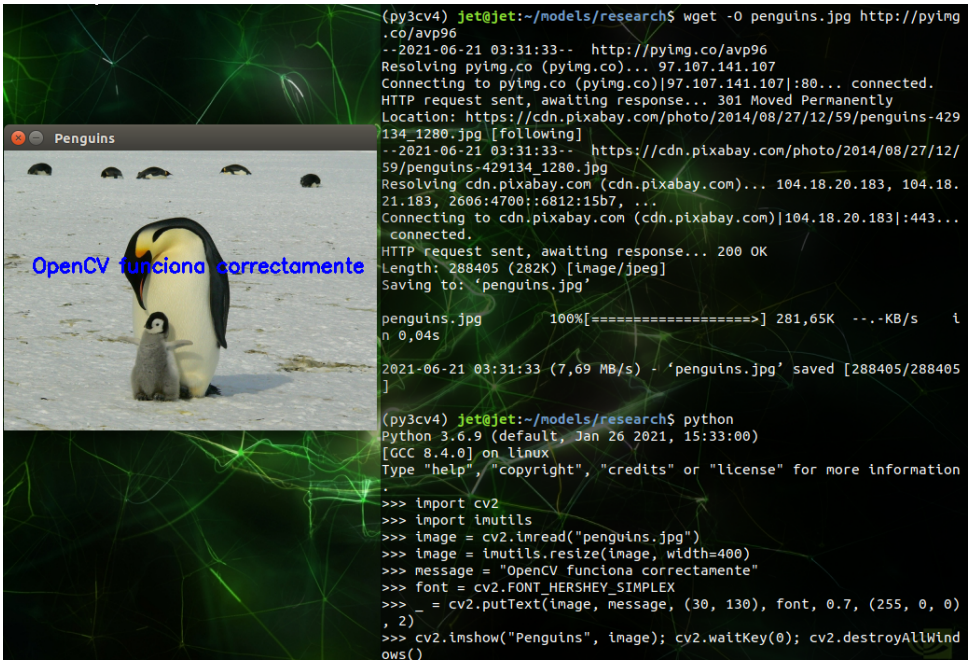


Figura 3.33 Captura de terminal y resultados del *test* de comprobación de OpenCV.

Por último, para probar la cámara de la Raspberry Pi en la Jetson Nano se hará uso de del proyecto `nanocamera`, el cual proporciona una interfaz simple que soporta cámaras CSI, USB, IP y RTSP. Puede ser instalado manualmente a través de su repositorio en GitHub, o mediante `pip`, siendo el segundo método el que se usará en este caso.

```
$ workon py3cv4
$ pip3 install nanocamera
```

Una vez instalada la interfaz se podrá probar la cámara mediante el archivo ejecutable de Python que se muestra a continuación, basado en un ejemplo de GitHub para probar una cámara CSI del propio repositorio de NanoCamera.

Código 3.1 test_camera.py

```
import cv2
import nanocamera as nano
import time

if __name__ == '__main__':
    # Create the Camera instance
    camera = nano.Camera(flip=0, width=640, height=480, fps=1)
    print('CSI Camera ready? - ', camera.isReady())
    while camera.isReady():
        try:
            # read the camera image
            frame = camera.read()
            # display the frame
            cv2.imshow("Video Frame", frame)
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
        except KeyboardInterrupt:
            break

    # close the camera instance
    camera.release()

    # remove camera object
    del camera
```

Ejecutando este archivo se obtendrá la imagen de la cámara si todo se ha realizado de forma correcta, como se ve en la Figura 3.34.

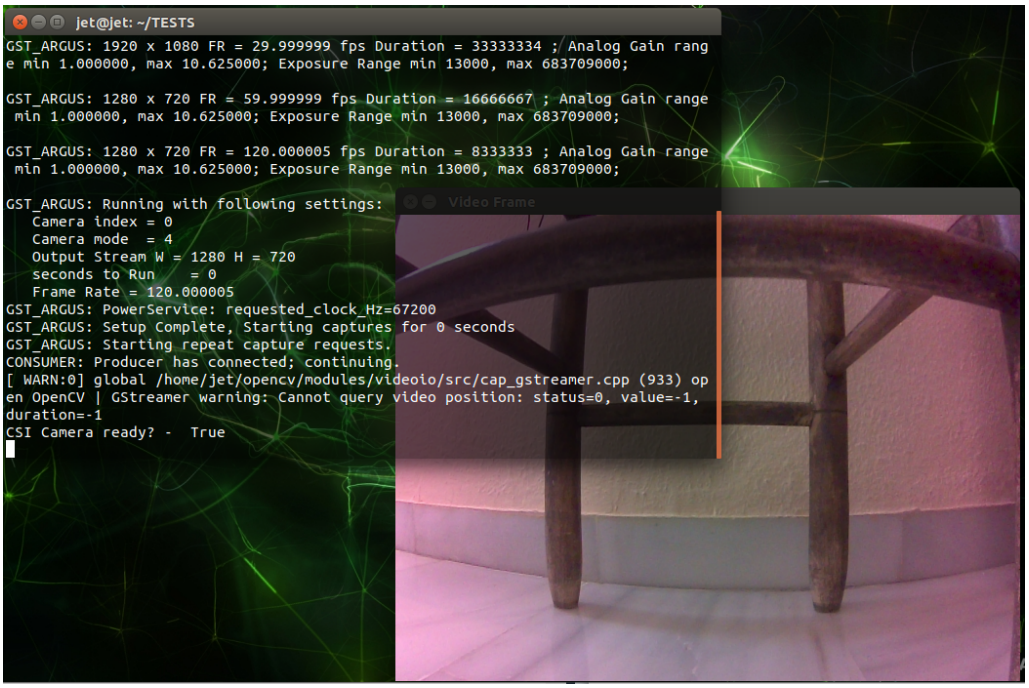


Figura 3.34 Captura de pantalla apreciando la prueba de la cámara CSI.

Finalizadas todas estas pruebas se pone de manifiesto que el conjunto de las instalaciones anteriores se completaron de forma adecuada y sin errores, y que todo funciona como es debido.

3.6.2 Instalación de YOLO

Para la instalación de YOLO en la Jetson Nano se usará la versión existente en el repositorio de Github de Alexey Bochkovskiy, la cual contiene YOLOv4 que será la designada para el proyecto, como se comentó en apartados anteriores. Con vistas a obtener un resultado estable y probado el proceso a realizar estará basado en el tutorial que se encuentra en [32], el cual es realizado para YOLOv3 pero los procedimientos son similares para YOLOv4. Antes de realizar la instalación, cabe destacar una aclaración, y es que YOLO es el algoritmo de detección de objetos el cual necesita de una red neuronal en la que poder correr, que en este caso se denomina *Darknet*. Ahora sí, se comenzará con los pasos de la instalacón.

Antes de realizar cualquier instalación siempre es buena práctica comprobar actualizaciones en los repositorios y en los paquetes.

```
$ sudo apt update
$ sudo apt upgrade
```

Tras esto, se exportará la ruta de CUDA (*Compute Unified Device Architecture*) que, como ya se explico anteriormente, se trata de una plataforma de programación y computación en paralelo desarrollada por Nvidia para ser usada en diferentes tareas que requieran sus GPUs.

```
$ export PATH=/usr/local/cuda-10.0/bin${PATH:+:${PATH}}
$ export LD_LIBRARY_PATH=/usr/local/cuda-10.0/lib64${LD_LIBRARY_PATH:+:${
    LD_LIBRARY_PATH}}
```

El siguiente paso lógico será la descarga de `darknet` desde el repositorio, así como los pesos de YOLOv4 para poder ejecutarlo.

```
$ git clone https://github.com/AlexeyAB/darknet
$ cd darknet
$ wget https://github.com/AlexeyAB/darknet/releases/download/
    darknet_yolo_v3_optimal/yolov4.weights
```

Para configurar que YOLO se ejecute con la GPU es necesario modificar el archivo `Makefile` con un editor de textos, como puede ser *Gedit*, y modificar los siguientes parámetros:

```
GPU=1
CUDNN=1
OPENCV=1
```

Con ellos se establece, como sus propios nombres indican, que la compilación se realice para ser usada con la GPU, con CUDNN y con OpenCV, por lo que ya solamente quedaría llamar a que comience a compilar.

```
$ sudo make
```

Después de una espera de varios minutos la compilación se habrá completado, por lo que se puede pasar a realizar pruebas del correcto funcionamiento de YOLO. Para ello se usará la foto de un perro para comprobar si es detectado, mediante el siguiente comando.

```
$ cd ~/darknet
$ ./darknet detector test cfg/coco.data cfg/yolov4.cfg yolov4.weights -ext_output
    dog.jpg
```

Para lo cual se obtendrá la salida de la Figura 3.35, pudiendo observar que efectivamente el resultado es correcto y predice que la probabilidad de ser un perro es del 98 %.

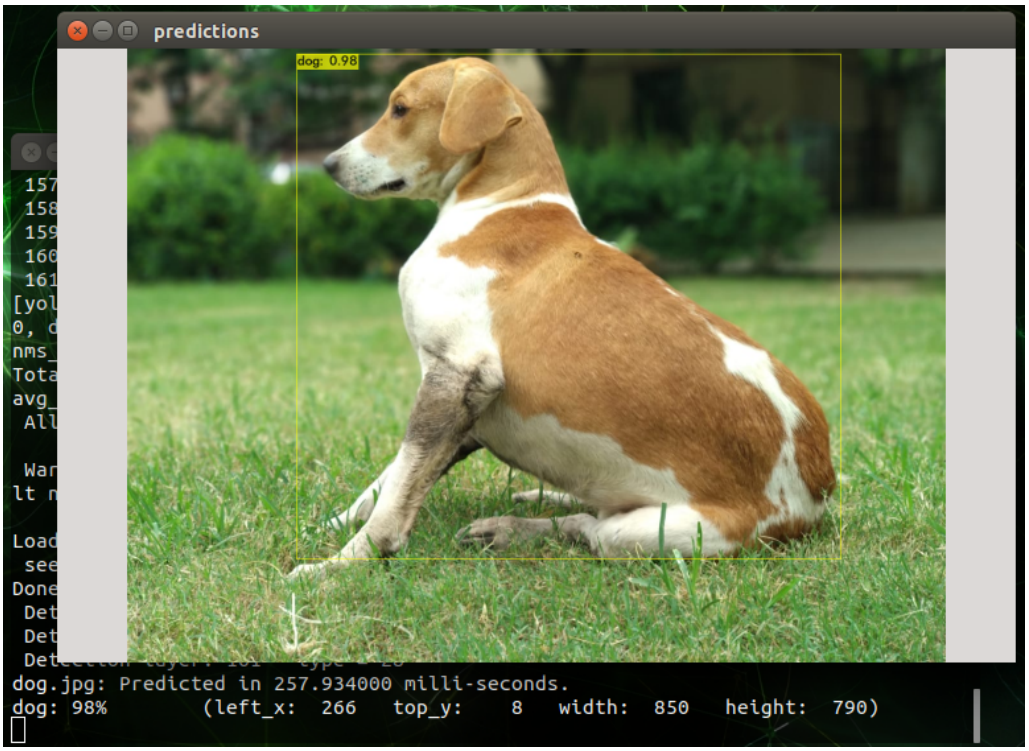


Figura 3.35 Captura de pantalla de la prueba de YOLOv4 con la foto de un perro.

Para realizar pruebas de vídeo con la cámara CSI el comando usado sería el siguiente:

```
$ ./darknet detector demo cfg/coco.data cfg/yolov4.cfg yolov4.weights -c 0
```

Indicando con `-c 0` que se usará la cámara del puerto 0. Esto abrirá una ventana similar a la de la Figura 3.35, pero con la diferencia de que la imagen se mostrará en tiempo real y se obtendrá a través de la cámara.

Inicialmente, con la configuración por defecto, el resultado que se obtuvo ofrecía una tasa de *frames* demasiado pobre, alrededor de 1 *fps*, lo que hacía insostenible detectar objetos en *streaming*. Esto fue posible mejorarlos modificando el fichero `yolov4.cfg`, ubicado en el directorio `/darknet/cfg`. Los valores por defecto que se procederá a modificar son:

```
batch= 64
subdivisions=8
width=608
height=608
```

Los cuales representan los lotes, divisiones y ancho y alto de las imágenes, por lo que al no ser una gráfica muy potente se debe bajar para obtener unos resultados más decentes, aunque pueda empeorar algo la detección. Se fueron modificando estos valores consiguiendo diferentes resultados: por ejemplo, si se reducía el tamaño de las imágenes a 256×256 se conseguía doblar los *fps* hasta una cifra alrededor de 2 *fps*, mientras que con un tamaño de 220×220 se alcanzaban los 3 *fps*. Se decidió para alcanzar aún una mayor velocidad reducir el tamaño del *batch* y de las subdivisiones, hasta el valor de 1, alcanzando así una cifra estable de en torno a 5 *fps*, un valor muy superior al inicial.

Con distintas pruebas se comprobó que la precisión de la detección no se veía comprometida de manera importante al cambiar estas variables, por lo que los valores finales se mantuvieron como se han explicado y se muestran a continuación.

```
batch= 1
subdivisions= 1
width= 220
height= 220
```

3.6.3 Entrenar YOLO para objetos propios

Para la realización de este proyecto era necesario entrenar YOLO con el *dataset* propio, el cual se desarrolla en el Capítulo 4, Dataset. Antes de poder realizar esto era necesaria la instalación de todos los recursos en un ordenador con la capacidad de cálculo necesaria para la realización del entrenamiento, ya si este proceso se realizara en la Jetson Nano el tiempo empleado sería extremadamente lento por su baja capacidad gráfica. Es por ello que se procedió a instalar todos los recursos necesarios en un portátil con una gráfica Nvidia GeForce GTX 950M, la cual no es tampoco demasiado potente, con una puntuación en *Compute Capability* según la página de Nvidia de CUDA de 5.0, pero cumplía lo imprescindible para poder ofrecer una experiencia de entrenamiento aceptable.

El sistema operativo instalado en el ordenador era Ubuntu 18.04, la misma versión que en la que se basa la versión de JetPack instalada en la Jetson, con la intención de que el sistema de entrenamiento fuera lo más similar posible al que posteriormente se usara para detección en tiempo real.

El primer paso que se realizó en el el ordenador fue la instalación de los *drivers* de Nvidia, añadiendo el repositorio de los controladores y después instalándolo como se ve a continuación, tras lo cual será necesario un reinicio del sistema.

```
$ sudo add-apt-repository ppa:graphics-drivers/ppa
$ sudo apt update
$ sudo ubuntu-drivers autoinstall
$ sudo reboot
```

Cuando se inicia de nuevo el sistema ya están corriendo los drivers de Nvidia, por lo que se puede proceder a realizar la instalación de CUDA. Al querer recrear las condiciones de la instalación de la Jetson Nano, como se puede apreciar en la Figura 3.30, consta de CUDA en su versión 10.0 y cuDNN en su versión 7.5.0.

Para instalar CUDA 10.0, la última versión de Ubuntu con soporte en la página de Nvidia es la 18.04, como se puede apreciar en la Figura 3.36, siendo esta otra de las razones por las que se decidió usar esta versión del sistema y no una posterior como podría haber sido la 20.04.

Figura 3.36 Captura de la *web* de Nvidia para la descarga de CUDA *Toolkit* 10.0

Para proceder a la instalación, por tanto, de CUDA 10.0 lo primero será descargar el archivo `.deb` desde la página mencionada. Lo segundo que se comprobó es la versión de los *drivers* de Nvidia instalada en el paso anterior, ya que según la tabla de la Figura 3.37 estos deben ser una versión igual o superior a la 410.48.

Table 3. CUDA Toolkit and Corresponding Driver Versions

CUDA Toolkit	Toolkit Driver Version	
	Linux x86_64 Driver Version	Windows x86_64 Driver Version
CUDA 11.3.1 Update 1	>=465.19.01	>=465.89
CUDA 11.3.0 GA	>=465.19.01	>=465.89
CUDA 11.2.2 Update 2	>=460.32.03	>=461.33
CUDA 11.2.1 Update 1	>=460.32.03	>=461.09
CUDA 11.2.0 GA	>=460.27.03	>=460.82
CUDA 11.1.1 Update 1	>=455.32	>=456.81
CUDA 11.1 GA	>=455.23	>=456.38
CUDA 11.0.3 Update 1	>= 450.51.06	>= 451.82
CUDA 11.0.2 GA	>= 450.51.05	>= 451.48
CUDA 11.0.1 RC	>= 450.36.06	>= 451.22
CUDA 10.2.89	>= 440.33	>= 441.22
CUDA 10.1 (10.1.105 general release, and updates)	>= 418.39	>= 418.96
CUDA 10.0.130	>= 410.48	>= 411.31

Figura 3.37 Captura de la *web* de documentación de CUDA *Toolkit*.

Esto puede ser comprobado rápidamente con:

```
$ cat /proc/driver/nvidia/version
```

Cuya salida fue:

```
NVRM version: NVIDIA UNIX x86_64 Kernel Module 465.19.01
```

Otra comprobación necesaria era saber si el compilador `gcc` estaba instalado, algo rápido también de conocer mediante:

```
$ gcc --version
```

Cuya salida fue, indicando la versión instalada:

```
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
```

Por lo que, una vez comprobado que todos los requisitos se cumplían, y que también CUDA 10.0 era compatible con la gráfica del ordenador, ya que requería una Compute Capability mínima de 3.0, se puede proceder a instalar el paquete descargado. Para ello, abriendo una terminal en el directorio de descarga del `.deb` se introducirá lo siguiente:

```
$ sudo dpkg -i cuda-repo-ubuntu1804_10.0.130-1_amd64.deb
$ sudo apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda
    /repos/ubuntu1804/x86_64/7fa2af80.pub
$ sudo apt-get update
$ sudo apt-get install cuda-10-0
```

Tras ello, se procede a descargar cuDNN *v*7.5.0 para CUDA 10.0 desde la página de desarrolladores de Nvidia, en la cual es necesario el registro para obtener enlace al archivo comprimido. Una vez finalizada la descarga, se descomprimirá el archivo y se abrirá una terminal en la carpeta obtenida tras descomprimir, donde se introducirán los siguientes comandos:

```
$ sudo cp -P cuda/targets/ppc64le-linux/include/cudnn.h /usr/local/cuda-10.0/
    include/
$ sudo cp -P cuda/targets/ppc64le-linux/lib/libcudnn* /usr/local/cuda-10.0/lib64/
$ sudo chmod a+r /usr/local/cuda-10.0/lib64/libcudnn*
```

Finalizado la copia de estos archivos solamente quedaría exportar la ruta del directorio de CUDA 10.0 y sus librerías al archivo `.bashrc`, para que se conozca cada vez que se abra la terminal.

```
$ echo 'export PATH=/usr/local/cuda-10.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-10.0/lib64:${LD_LIBRARY_PATH:+:${
    LD_LIBRARY_PATH}}' >> ~/.bashrc

$ source ~/.bashrc
```

Conseguida la instalación de todo lo referido a la gráfica, tocaba realizar el procedimiento de instalar los paquetes y librerías de Python tal como se realizó en la Jetson, respetando las versiones de cada uno de ellos para evitar incompatibilidades en el archivo de pesos que posteriormente se usaría. Muestra de estas instalaciones serían *Virtualenv*, *Keras*, *TensorFlow*, *OpenCV* o *Numpy* entre otros.

Una vez completada la instalación de todo lo necesario para ejecutar YOLO, se procedió a su descarga, tal como se realizó para la Nano, comprobando que las pruebas realizadas eran correctas, por lo cual se podía comenzar a preparar el entrenamiento de la red para el proyecto.

Para reentrenar la red de YOLO se seguirá el tutorial del repositorio oficial de YOLO en GitHub con la intención de conseguir que detecte objetos personalizados. Por tanto, se comenzará con la descarga de de los pesos preentrenados `yolov4.conv.137`, el cual ubicaremos en el directorio `/darknet/build/darknet/x64` para ser utilizado más tarde al comienzo del entrenamiento.

Tras esto se realizará una copia del fichero `yolo-customv4.cfg` ubicado en `/darknet/cfg`, siendo el nombre elegido para la copia `yolo-tfg.cfg`. En él deben realizarse varios cambios mediante un editor de texto:

- `Batch=64` (línea 6)
- `subdivisions=14` (línea 7). A pesar de ser esta la recomendación del tutorial, finalmente se tuvo que poner un valor de `subdivisions=64`, ya que las bajas capacidades de memoria de la tarjeta gráfica hacía que no fuera posible entrenar con un valor tan bajo, quedándose sin memoria y paralizando el inicio del entrenamiento.
- `max_batches=classes * 2000` (línea 20), pero no menos que el número de imágenes de entrenamiento ni menos que 6000, por ello se estableció `max_batches = 12000`, ya que hay 6 clases.
- `steps=9600,10800` (línea 22), ya que debía tener valores del 80 % y 90 % de `max_batches`.
- `width=416 height=416` (líneas 8 y 9) o valores múltiplos de 32. Por la misma razón que anteriormente se tuvo que reducir estas dimensiones hasta un valor de `width=224 height=224`.
- `classes=6` (líneas 970, 1058 y 1146), ya que se tienen 6 clases diferentes en el *dataset*, como ya se verá en el 4.
- `filters=(classes + 5) * 3` antes de cada capa convolucional (línea 963, 1051 y 1139), por lo que quedaría `filters=33`.

Terminada la edición del fichero `.cfg` se pasará a la creación de `tfg.names` en el directorio `/darknet/build/darknet/x64/data`. El archivo constará de los seis nombres de los objetos a detectar, como se muestra a continuación:

```
azul
gris
naranja
rojoBig
rojoBigrayado
rojoSmall
```

Otro fichero que será necesario crear es `tfg.data`, ubicado en el mismo directorio que el anterior. En él se debe indicar el número de clases, la ubicación del archivo que contenga la ruta de cada una de las imágenes usadas para entrenamiento, la ruta del archivo `tfg.names` realizado antes, así como la carpeta en la que se irán guardando los *backups* de los archivos de pesos durante el entrenamiento, quedando todo esto de la siguiente manera:

```
classes= 6
train = build/darknet/x64/data/tfg_train.txt
valid = build/darknet/x64/data/tfg_test.txt
names = build/darknet/x64/data/tfg.names
backup = backup/
```

El último archivo de texto a crear será `tfg_train.txt`, que como se mencionaba anteriormente contiene la ruta de todas las imágenes usadas para el dataset. Una forma rápida de obtenerlas es mediante la terminal y el comando `ls`, el cual lista todos los elementos de la ruta que se le introduzca. Se puede configurar para que muestre sólo archivos de determinada extensión, como `.jpg` en este caso, y permite exportar la lista a un archivo de texto. Las imágenes están ubicadas en la ruta `/darknet/build/darknet/x64/data/tfg`, dentro de la cual se encuentran a su vez separadas en carpetas según el objeto que aparezca en las imágenes.

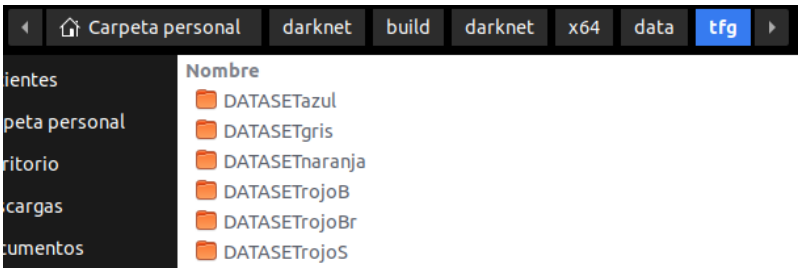


Figura 3.38 Captura de pantalla del directorio donde se encuentra ubicado el *dataset*.

```
$ ls build/darknet/x64/data/tfg/DATASETazul/*.jpg >> tfg_training.txt
$ ls build/darknet/x64/data/tfg/DATASETgris/*.jpg >> data/tfg_training.txt
$ ls build/darknet/x64/data/tfg/DATASETnaranja/*.jpg >> data/tfg_training.txt
$ ls build/darknet/x64/data/tfg/DATASETrojoB/*.jpg >> data/tfg_training.txt
$ ls build/darknet/x64/data/tfg/DATASETrojoBr/*.jpg >> data/tfg_training.txt
$ ls build/darknet/x64/data/tfgDATASETrojoS/*.jpg >> data/tfg_training.txt
```

Para la realización de las *bounding boxes* del *dataset* fue necesaria la instalación y uso del programa Yolo_mark, el cual permite etiquetar cada imagen individualmente mediante una sencilla pero eficaz interfaz gráfica. Es el programa que se recomienda en el mismo tutorial que se está siguiendo, y se encuentra también en el repositorio de GitHub de Alexey. Se mostrará en profundidad en el siguiente Capítulo, en el apartado 4.4.

Las imágenes destinadas a validación también serán colocadas en su ubicación correcta, concretamente en la carpeta `/darknet/build/darknet/x64/data/tfg_valid`, así como el archivo de texto `tfg_valid.txt` incuirá la ruta de cada una de las imágenes destinadas este fin.

Tras tener todos los ficheros de texto en sus respectivas ubicaciones, el *dataset* preparado y etiquetado, y los pesos iniciales de YOLOv4, se puede comenzar el proceso de entrenamiento introduciendo la siguiente línea de comando:

```
./darknet detector train build/darknet/x64/data/tfg.data cfg/yolo-tfg.cfg yolov4.
    conv.137
```

Esto dará comienzo a la etapa de entrenamiento, abriéndose una interfaz gráfica en la que se puede ir observando el *loss* actual medio en cada iteración, así como el número de repeticiones y un tiempo restante estimado. Esta gráfica será mostrada junto con más resultados en el Capítulo 5

Para asegurar que *darknet* se está ejecutando con la GPU y no con la CPU, lo que sería mucho más lento, se puede introducir el siguiente comando:

```
$ nvidia-smi -l
```

El cual muestra en tiempo real, ya que se le ha indicado que entre en *loop* con `-l`, los procesos que se están ejecutando con la tarjeta gráfica, y como se puede apreciar en la Figura 3.39, señalado en rojo está el proceso *darknet*. Algo destacable, señalado en azul, es el consumo de memoria que conlleva el entrenamiento, estando prácticamente al límite de los 2 gigas de los que dispone la GPU del PC. Esto refuerza el argumento de que era necesario bajar el ancho y alto de las imágenes de entrenamiento para poder ejecutarlo.

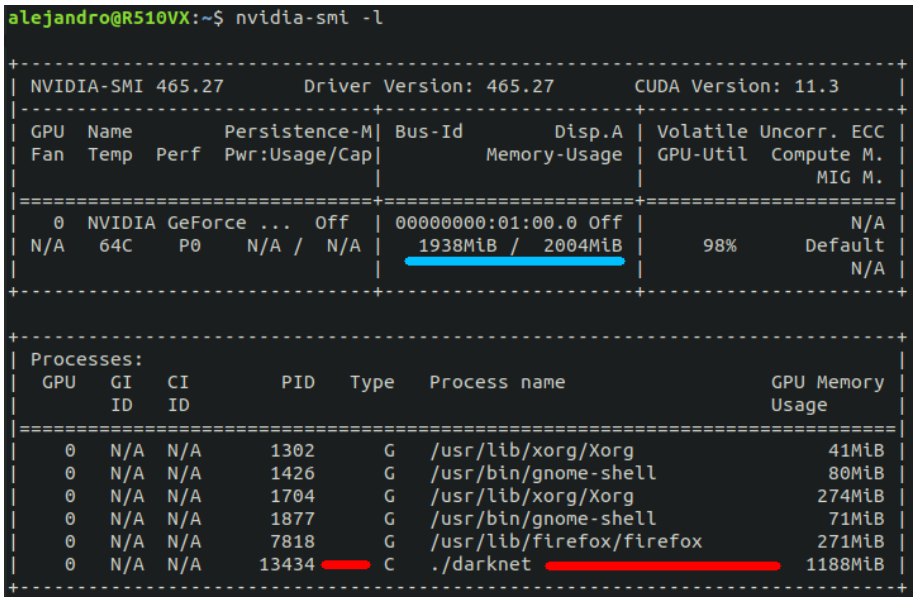


Figura 3.39 Captura de pantalla de la terminal mostrando los procesos que usan la GPU.

Si se necesita interrumpir el entrenamiento en algún momento, los pesos serán guardados en la carpeta `darknet/backup`, pudiendo retomar el entrenamiento a partir de ellos, cambiando el comando anterior de la siguiente manera:

```
$ ./darknet detector train build/darknet/x64/data/tfg.data cfg/yolo-tfg.cfg backup
    /yolo-tfg_last.weights
```

Una vez se considere finalizado el entrenamiento, los pesos definitivos se encontrarán en la carpeta mencionada, pudiendo hacer uso de ellos en lugar de los de YOLOv4 descargados. Esto será explicado de forma más detallada cuando se muestren los resultados en los siguientes capítulos.

4 Dataset

"The word pretty is unworthy of everything you will be. And no child of mine will be contained in five letters. You will be pretty intelligent, pretty creative, pretty amazing. But you will never be merely pretty."

—Kattie Makkai

Para la realización del dataset se han tenido en cuenta principalmente dos tipos de factores enfocados en software y en hardware.

4.1 *Hardware*

Figura 4.1 Intel AC 8265.[1]

En primer lugar era necesario instalar la Jetson Nano en el JetBot mediante los cuatro tornillos de anclaje. La cámara también se colocó en el soporte destinado para ella, mediante el cual puede ser fijada con facilidad, o movida verticalmente para acomodar la posición de la vista de la misma según las necesidades. Además, para una mayor libertad se le instaló en el *slot* M.2 el adaptador *Intel Dual Band Wireless-AC 8265* que permitía conectar la Jetson Nano mediante wifi y poder controlarla remotamente sin necesidad de cable ethernet. Para poder acceder al puerto M.2 fue necesario desmontar el disipador ligado a la Jetson Nano, ya que el *slot* se encontraba debajo de este. Una vez emplazado el adaptador se volvió a atornillar el disipador de forma correcta. El JetBot dispone de dos grandes antenas que irían directamente conectadas con la AC 8265 para tener una buena conexión inalámbrica, las cuales son apreciables en la Figura 4.2.

[1] Fuente: *https://images-na.ssl-images-amazon.com/images/I/71VUI2Q1ABL._AC_SY679_.jpg*

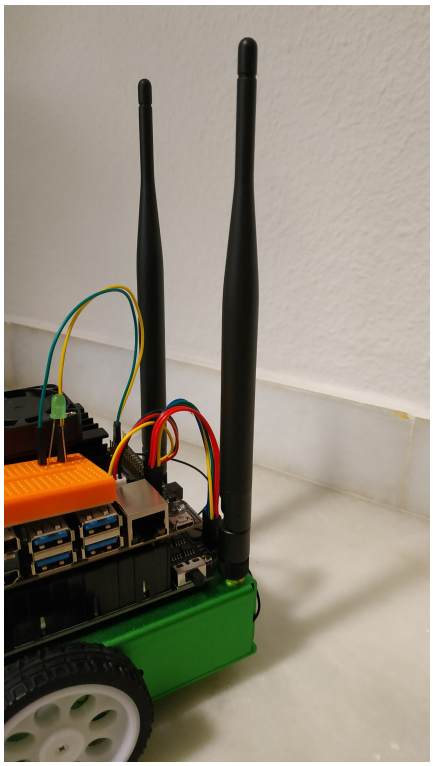


Figura 4.2 Antenas del JetBot.

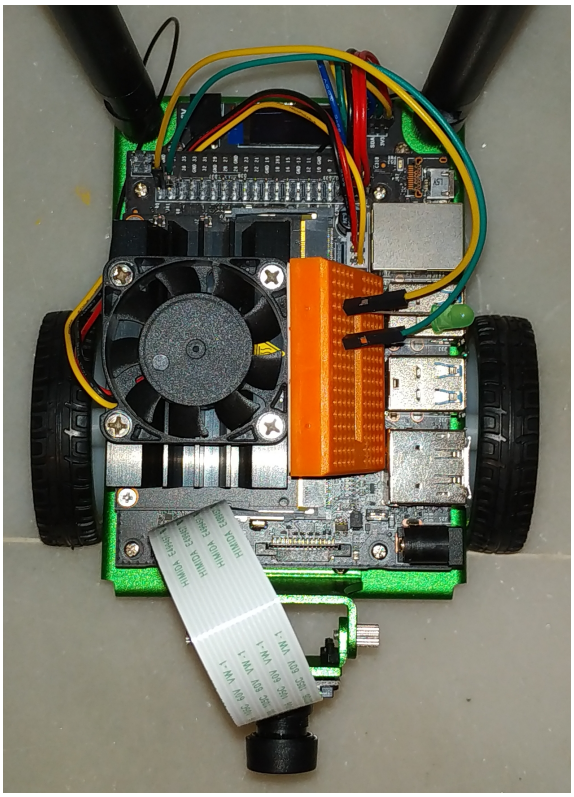


Figura 4.3 Ventilador acoplado al disipador.

Con la finalidad de alcanzar una mejora en el rendimiento se consideró también la instalación del mini ventilador incluido en el kit del JetBot para proporcionar una mejor refrigeración al módulo,

el cual iría atornillado directamente sobre los orificios destinados a ellos ubicados en el disipador, pudiendo observarse con claridad en la Figura 4.3.

Para facilitar la creación del *dataset*, se incluyó un led verde en una mini *protoboard* conectado entre la GND y unos de los pines accesibles a escritura (pin 26) de la Jetson, con la intención de que se encendiera cuando se fuera a realizar una fotografía. Sería algo similar a un *flash*, pero en lugar de iluminar la imagen su funcionalidad consistiría en avisar a la persona que está realizando el banco de datos de cuándo se ha tomado la fotografía. Este también es apreciable en la Figura 4.3, colocado en una posición correcta para poder ser visualizada su iluminación desde arriba por la persona que esté manipulando el vehículo. Se verá algo más a fondo su funcionamiento y desarrollo en el apartado de *Software* (4.2).

4.2 Software

En la sección de software se decidió realizar un *script* en *python* para facilitar la obtención de imágenes de una forma sencilla y continua. En primer lugar se instaló la interfaz de cámara de *python* para la Jetson Nano llamada NanoCamera. Se trata de una interfaz creada por el usuario *thehapyone* en Github, la cual da soporte a cámaras USB y CSI. Se puede instalar de manera fácil a través de *pip*, o manualmente descargándola desde el repositorio oficial[2].

La resolución de las imágenes tomadas era un parámetro a tener en cuenta para la realización del *dataset*, así como para el uso de la NanoCamera, por lo que se decidió que fuera una resolución relativamente reducida dadas las características recortadas del sistema, optando finalmente por imágenes de 640×480 píxeles.

Para controlar el led mencionado en el apartado anterior fue necesaria la instalación de la librería de *python* `Jetson.GPIO`, con el propósito de manipular las tensiones de los pines de entrada y salida de propósito general de la placa. Se decidió que se tomaría una imagen por segundo, es decir, 1 *fps*, por lo que el led sería encendido cuando se realizara la fotografía, estando iluminado durante medio segundo, para después apagarse durante otro medio segundo y repetir el proceso. La velocidad de captura fue determinada así para dar tiempo a variar el entorno a medida que se iban realizando imágenes y ser capaz de realizar un banco de datos variado.

También con el propósito de parar la captura de imágenes en cualquier momento deseado y poder retomarla en un instante posterior se incorporó la detección de una interrupción con el comando `Control + C`, y un *handler* que se encargara de llevar a cabo la paralización e informar al usuario de ello.

Las imágenes tomadas serían nombradas siguiendo la terminología `dataset_00000.jpg` en adelante, guardándose en la carpeta `DATASET` destinada a ello, dentro de la propia carpeta de usuario, y aparecerá un mensaje por pantalla cada vez que se almacene un nuevo archivo.

El resultado final del código en *python* quedaría de la siguiente forma:

Código 4.1 doing_dataset.py

```
# import the necessary packages
import nanocamera as nano
import RPi.GPIO as gpio
import imutils
import time
```

[2] *https://github.com/thehapyone/NanoCamera*

```
import cv2
import signal

keep_alive = False
def handler(signal, frame):
    global keep_alive

    keep_alive ^= True

signal.signal(signal.SIGINT, handler)

if __name__ == '__main__':
    camera = nano.Camera(flip=0, width=640, height=480, fps=1)
    pin_num = 26
    gpio.setmode(gpio.BCM)

    # Set GPIO LED
    gpio.setup(pin_num,gpio.OUT, initial = gpio.HIGH)
    time.sleep(2.0)

    # loop over frames
    enter = input("Press enter to start recollecting images \n")
    n= 0

    while True:
        while camera.isReady():
            if keep_alive:
                gpio.output(pin_num,gpio.LOW)
                time.sleep(0.5)
                gpio.output(pin_num,gpio.HIGH)
                frame = camera.read()
                cv2.imshow("Frame", frame)
                cv2.waitKey(1)
                # FILENAME
                filename = 'dataset_' + str(n).zfill(5) + '.jpg'
                n += 1
                # SAVING THE IMAGE
                cv2.imwrite('/home/jet/DATASET/' + filename, frame)
                print('Successfully saved ' + filename + '\n')
                time.sleep(0.5)
            else:
                print('STOPPED!')
                time.sleep(1)

    # release the video stream and close open windows
    print("End!")
    gpio.cleanup()
    camera.release()
    del camera
    cv2.destroyAllWindows()
```

4.3 Primer *dataset*

En primer lugar se comenzó realizando un *dataset* en el laboratorio con cajas de diferentes tamaños pero todas del mismo color, rojas. Se realizó capturando una imagen por segundo con el *script* mencionado en el apartado anterior. A pesar de obtener un *dataset* polivalente, se consideró que no era el adecuado para este trabajo, ya que lo que se pretendía era detectar y clasificar los obstáculos, por lo que con obstáculos tan similares no sería posible diferenciarlos entre ellos y, consecuentemente, no sería posible su clasificación, sino únicamente afirmar o negar la presencia de obstáculos en la imagen. Es por ello que se realizaron dos baterías de imágenes partiendo de ese concepto, una del espacio del laboratorio donde se realizó el *dataset* sin la presencia de obstáculos en la imagen, y otra batería con presencia de obstáculos, pudiendo ser un sólo obstáculo individual, o multitud de ellos repartidos por el espacio.

Figura 4.4 Ejemplo de imagen del *dataset* con obstáculos.

En esta primera imagen de la Figura 4.4 se pueden apreciar cuatro diferentes cajas rojas en el espacio del laboratorio, pudiendo ser una tarea fácil detectar su presencia mediante una red neuronal, pero complicado de identificar cuál sería cada una de las cajas mediante una red neuronal. Podría quizás ser necesario añadir un sensor de proximidad o un lídar para saber a qué distancia de ella está el robot, y cuál es su tamaño, para así poder clasificarlas de una forma más adecuada que usando exclusivamente visión por cámara.

Las condiciones de luz de las fotografías consistían en únicamente luz artificial, ya que el laboratorio no disponía de ventanas, siendo una manera de simplificar brillos, sombras, y demás desajustes que puede ocasionar la luz natural en el entorno de trabajo de detección de obstáculos por visión artificial, facilitando de este modo también el entrenamiento y siendo necesario un menor número de imágenes para conseguir unos resultados correctos, ya que las variables son menores.

Figura 4.5 Ejemplo de imagen del *dataset* sin obstáculos.

En esta Figura 4.5 se puede apreciar el entorno sin ningún obstáculo alrededor como ya se ha comentado. A pesar de ser un buen *dataset*, no se llegó a usar para entrenar ninguna red neuronal dada su inutilidad en este proyecto.

4.4 Segundo *dataset*

Para la realización del segundo *dataset* se tuvieron ya en cuenta los errores cometidos en la realización del primero, estudiando mejor las opciones antes de ponerse manos a la obra, y se decidió la elección de seis cajas distintas. Las cajas fueros forradas con cuatro papeles de colores distinto, una azul, una gris, una naranja, y tres rojas. De las tres rojas, dos tenían aproximadamente el mismo tamaño, y a una de esas dos se le pintaron rayas negras diagonales en todas sus caras para diferenciarla de la lisa, con la intención de que a una distancia lejana fuera difícil de distinguir, sobre todo debido a la baja calidad de la cámara utilizada para las fotos. La última caja roja tendría un tamaño más reducido, pero en diferentes ocasiones, sobre todo a una distancia muy cercana, también puede ser confundida con las cajas grandes, por lo que así se aumentó la dificultad de la identificación de los obstáculos.

Figura 4.6 Cajas designadas como obstáculos para el proyecto.

La realización de este *dataset* no fue realizado en el laboratorio como el anterior, para ahorrar dificultades de disponibilidad del sitio ya que suele estar ocupado por más personas.

Se decidió realizar el banco de imágenes en la cocina de casa, siendo esta una habitación con persianas para poder disminuir la incidencia externa de la luz solar al mínimo, y depender exclusivamente de luz artificial, consiguiendo así una menor variabilidad en las situaciones y, por lo tanto, menos imágenes necesarias para entrenar.

Debido a la baja calidad de la cámara las imágenes del *dataset* no ofrecen un gran detalle, por lo que es difícil distinguir la caja roja lisa de la caja roja rayada en determinadas situaciones de luz o de distancia.

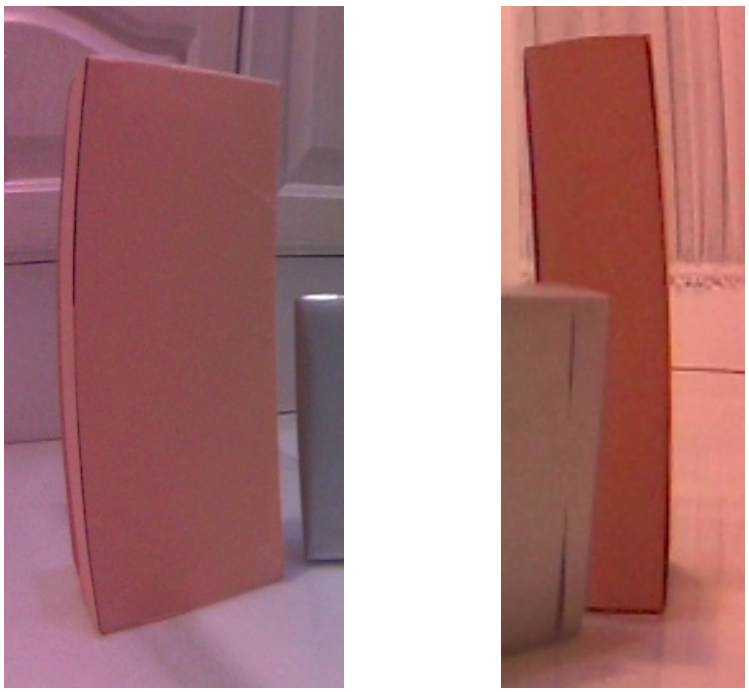


Figura 4.7 Ejemplo de imágenes de la caja naranja que pueden ser confundidas con la roja por el tono rojizo de la imagen.

Además, la cámara introduce en la imagen un color rojizo que hace que a veces la caja naranja parezca roja, y otras veces que las rojas parezcan naranja. Esto no suele afectar a las cajas azul y gris ya que sus colores son bastante más independientes.

En la Figura 4.7 puede verse esta deficiencia en la imagen, la cual a ojo humano puede no ser muy dramática la similitud entre ambos colores, pero para en una red neuronal es posible que según la incidencia de la luz determine que se trata de una caja que no es, cambiando totalmente su acierto.

A pesar de ellos el resultado de las imágenes se considera adecuado para la realización del proyecto, sirviendo para entrenar de manera correcta una red neuronal. Muestra de cada una de las cajas en imágenes del *dataset* se puede ver en la Figura 4.8.

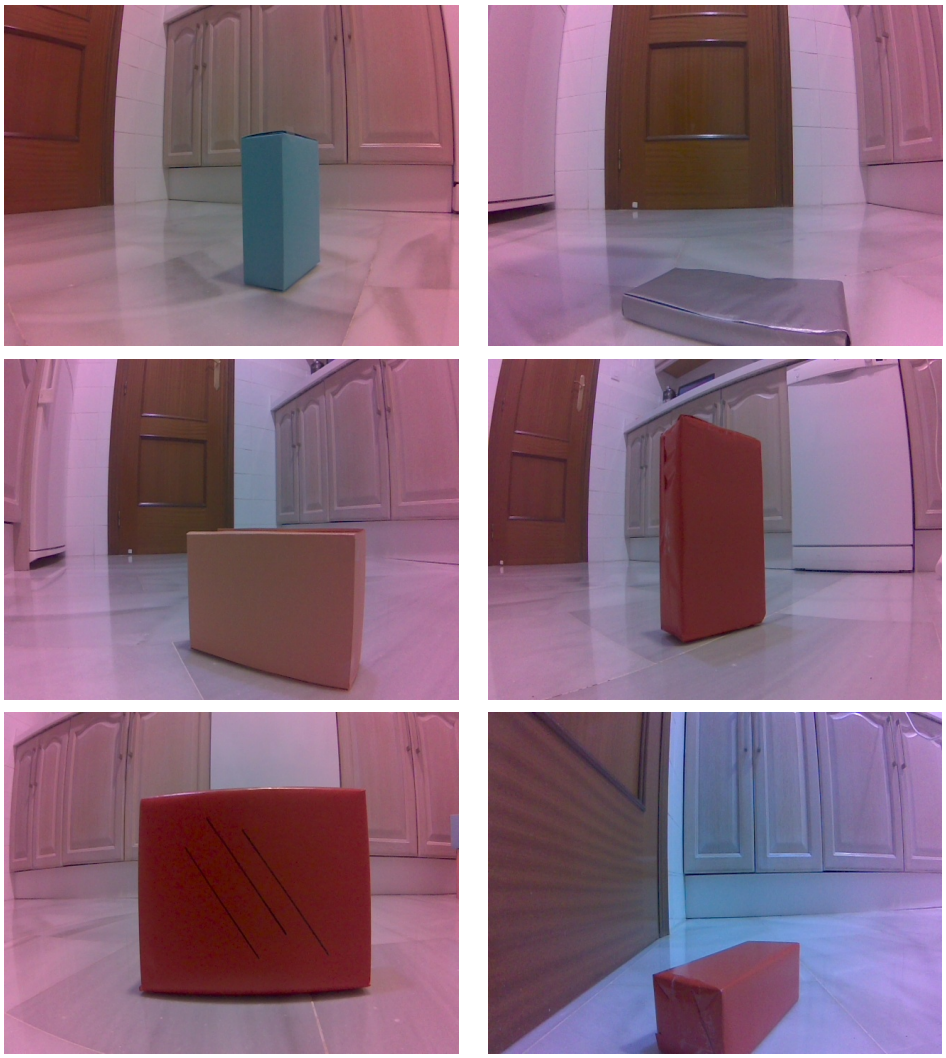


Figura 4.8 Ejemplo de imágenes del *dataset* de cada una de las cajas.

Para poder usar estas imágenes para el entrenamiento de YOLO es necesaria la realización de las *bounding boxes* en cada una de ellas. Esto se realizará mediante uso del programa *Yolo_mark*, el cual permite etiquetar cada una de las imágenes de forma rápida y sencilla mediante una interfaz gráfica bastante simple pero intuitiva.

El *software* puede ser instalado siguiendo las instrucciones de su repositorio oficial en GitHub. En primer lugar será necesario clonar el repositorio a la carpeta donde se desee, por ejemplo la

carpeta personal del usuario.

```
$ cd ~
$ git clone https://github.com/AlexeyAB/Yolo_mark
```

A continuación, se cambiará de directorio la terminal a la carpeta descargada mediante el comando `git`, y se compilarán los archivos mediante `CMake`.

```
$ cd Yolo_mark/
$ cmake .
$ make
```

Una vez finalizado, solamente quedará iniciarlo como un archivo ejecutable.

```
$ . linux_mark.sh
```

Esto abrirá la interfaz gráfica del programa con unas imágenes de muestra, tal como se ve en la Figura 4.9, las cuales muestran varias vistas de objetos aéreos como aviones o pájaros, los cuales están etiquetados y recuadrados como demostración de las capacidades del programa.

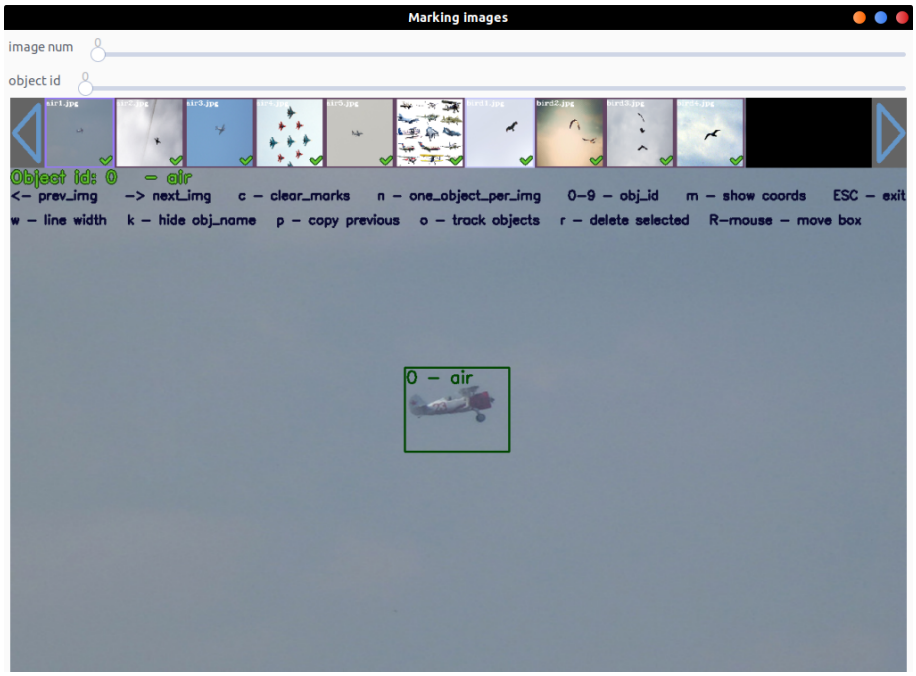


Figura 4.9 Primer inicio de *Yolo_mark*.

Para clasificar imágenes del *dataset* particular del proyecto se pasará a eliminar los ficheros de muestra que se encuentran en la ubicación `Yolo_mark/x64/Release/data/img`, y se copiarán en él las fotos propias para realizar su *bounding box*.

El siguiente paso una vez copiadas las imágenes en la ruta adecuada será modificar el archivo de texto `Yolo_mark/x64/Release/data/obj.data`, en el cual se sustituirá el valor por defecto de las clases por el número de objetos diferentes que tenga el *dataset* propio, en este caso 6. El resto del fichero se dejará tal cual estaba.

```
classes = 6
train = data/train.txt
valid = data/train.txt
names = data/obj.names
backup = backup/
```

Como se indica en el archivo anterior, los nombres de los diferentes objetos se encuentran en el archivo `Yolo_mark/x64/Release/data/obj.names`, por lo que se debe modificar introduciendo el nombre de cada objeto en una línea independiente, siguiendo el mismo orden que luego se usa para entrenar con YOLO, consiguiendo así que aparezcan estas etiquetas a la hora de ejecutar la interfaz gráfica.

```
azul
gris
naranja
rojoBig
rojoBigrayado
rojoSmall
```

Una vez realizados todos los pasos, se cargará de nuevo la interfaz gráfica a partir de su ejecutable `linux_mark.sh` y aparecerá la primera imagen del dataset. Para realizar el *bounding box* simplemente habría que seleccionar el recuadro con el ratón sobre el objeto que se desea etiquetar. La barra superior cambia entre las fotos del *dataset*, mientras que la segunda sirve para seleccionar la etiqueta que se desea asignar al recuadro que se realice. Si se pulsa la tecla `h` se abre un abanico de atajos de teclado, los cuales son:

- `←` : moverse hacia la izquierda en la barra seleccionada.
- `→` : moverse hacia la derecha en la barra seleccionada.
- `c` : eliminar las *bounding boxes* de la imagen actual.
- `n` : activa el modo en el que al realizar una *bounding box* se pasa automáticamente a la imagen siguiente.
- `0-9` : mueve automáticamente la barra de etiquetas al número indicado.
- `m` : muestra las coordenadas de la *bounding box*.
- `ESC` : cierra la interfaz gráfica y sale del programa.
- `w` : alterna entre diferentes grosores de la línea del recuadro.
- `k` : oculta el nombre del objeto en la *bounding box*.
- `p` : copia las *bounding boxes* de la imagen actual en la siguiente.
- `o` : sigue al objeto seleccionado con la *bounding box*.
- `r` : elimina la *bounding box* sobre la que se encuentra el cursor.
- `Clic derecho` : mueve la *bounding box* sobre la que se encuentra el cursor.

Todo lo explicado anteriormente puede ser apreciado en la Figura 4.10, donde se muestra una imagen del entorno gráfico del programa con una foto del *dataset* etiquetada.

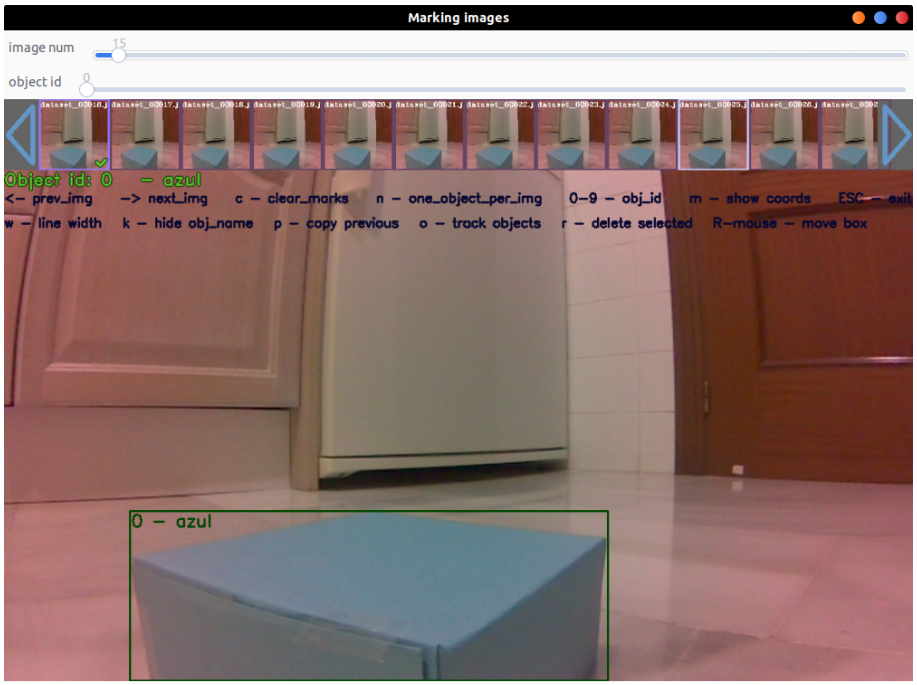


Figura 4.10 *Yolo_mark* durante el etiquetado de imágenes del *dataset*.

Cuando una imagen es etiquetada, el resultado se guarda en un archivo `.txt` con el mismo nombre que la foto, sólo que con diferente extensión. Dentro de este archivo la terminología seguida es la siguiente:

```
<object-class> <x_center> <y_center> <width> <height>
```

Es decir, cada *bounding box* se guarda en una línea diferente del archivo, dentro de la cual hay información separadas por espacios. La primera columna indica la clase que representa la *bounding box* de la línea actual, la segunda y tercera representan el centro del recuadro en el eje x e y respectivamente, y la cuarta y quinta apuntan el ancho y el alto del mismo.

Un ejemplo de un fichero de texto terminado sería el siguiente:

```
2 0.549219 0.664583 0.157813 0.331944
0 0.207031 0.545139 0.060938 0.143056
1 0.341797 0.598611 0.105469 0.030556
3 0.255078 0.496528 0.078906 0.206944
4 0.164062 0.494444 0.081250 0.230556
5 0.126953 0.577083 0.096094 0.070833
```

En él se encuentran seis líneas que representan las seis diferentes *bounding boxes* de la imagen que se muestra en la Figura 4.11.

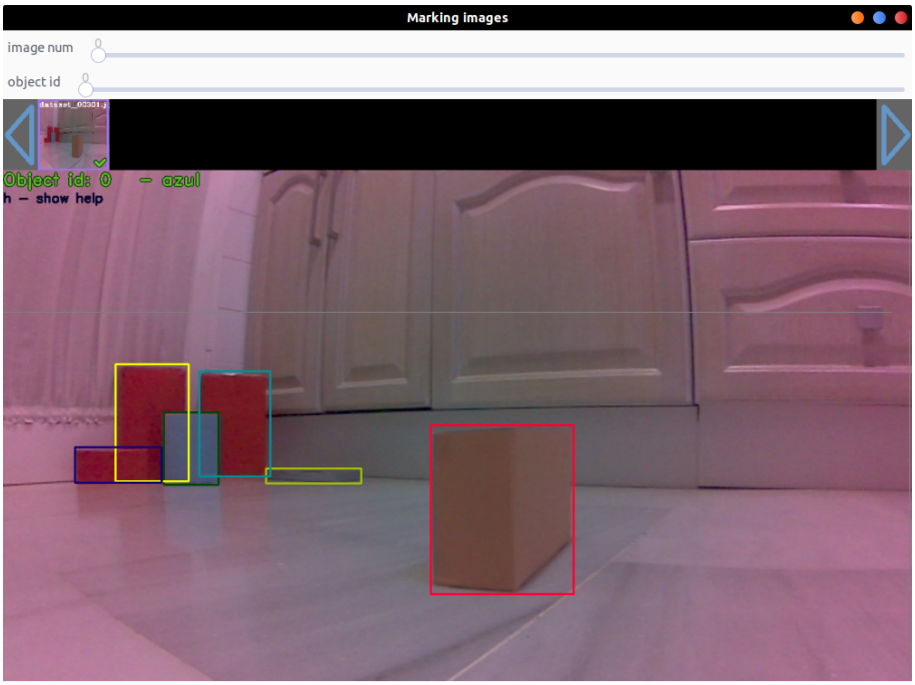


Figura 4.11 *Bounding boxes* de una imagen del *dataset*.

Una vez realizado esto para cada una de las fotografías del banco de imágenes, estas deben ser colocadas en la ruta correcta para poder comenzar el entrenamiento con YOLO, cuyos resultados se muestran en el siguiente Capítulo. El número de imágenes totales obtenidas para entrenamiento son 8.210.

Figura 4.12 Ejemplo de una imagen obtenida a partir del segundo vídeo realizado y destinada a validación de la red.

Tras la toma de fotografías, se realizaron dos vídeos en los cuales todas las cajas eran mostradas sin ninguna distinción entre ellas, para después poder ser empleados durante el proceso de *test*. El primer vídeo fue asignado para probar la eficacia de la red directamente sobre el vídeo, mientras que del segundo vídeo se extrajeron 200 imágenes para ser etiquetadas manualmente con las *bounding boxes* caja con la finalidad de poder obtener datos de validación a partir de ellas.

En estas imágenes las cajas se encuentran en ocasiones sin separación o apiladas unas sobre otras con la finalidad de poder poner a prueba la red neuronal en situaciones en las que se había visto envuelta durante el entrenamiento. Una muestra de este tipo de imágenes explicadas puede ser apreciada en la Figura 4.12.

5 Resultados

"En la investigación es incluso más importante el proceso que el logro mismo."

—Emilio Muñoz

Tras realizar el entrenamiento de la red neuronal YOLOv4 y la obtención de los parámetros y pesos finales del proyecto, deben realizarse varias comprobaciones para descubrir si los resultados obtenidos han sido los adecuados, siendo esto el tema principal que se tratará en este capítulo.

5.1 Proceso de entrenamiento

El procedimiento de entrenamiento comprendió un extenso periodo de tiempo durante el cual se estuvieron repitiendo iteraciones de forma indefinida, con la única realimentación de información que la aportada por las líneas de la terminal de comandos y la ventana gráfica que se abre durante el entrenamiento. Se comenzó inicialmente con 8.210 imágenes, pero debido a un fallo en el proceso durante el trazado manual de las *bounding boxes* este número era contradictorio, ya que había fotografías con objetos no clasificados, lo que producía una inestabilidad en los resultados, y haciendo que el error no convergiera. Como se puede apreciar en la Figura 5.1, la *loss* comenzó desde un valor muy elevado, ya que se partía de los pesos de YOLOv4, los cuales nunca habían clasificado imágenes con este tipo de cajas. La función de pérdida o *loss* es un concepto clave en el *deep learning*, la cual evalúa la desviación entre las predicciones realizadas por la red y los valores reales de los datos [64]. Es decir, la *loss* indica cómo de mala ha sido una predicción en un ejemplo concreto, y la función de *loss* se encarga de recopilar los valores individuales obtenidos de cada cada entrada al modelo durante el proceso de entrenamiento [63]. Cuanto más pequeño sea el valor de la *loss* más correcta será la predicción realizada por el modelo, siendo cero únicamente en el caso en el que la predicción sea perfecta.

Este primer intento duró alrededor de 10 horas de entrenamiento, lo cual era aún insuficiente dado su alto valor de *loss* media actual de 0.9768. Esto era, en parte, debido al error mencionado, el cual no se detectó hasta más adelante. La red siguió siendo entrenada de esta misma forma aproximadamente durante 24 horas más, con varias interrupciones ya que no era posible mantener el ordenador encendido realizando únicamente este proceso durante tanto tiempo.

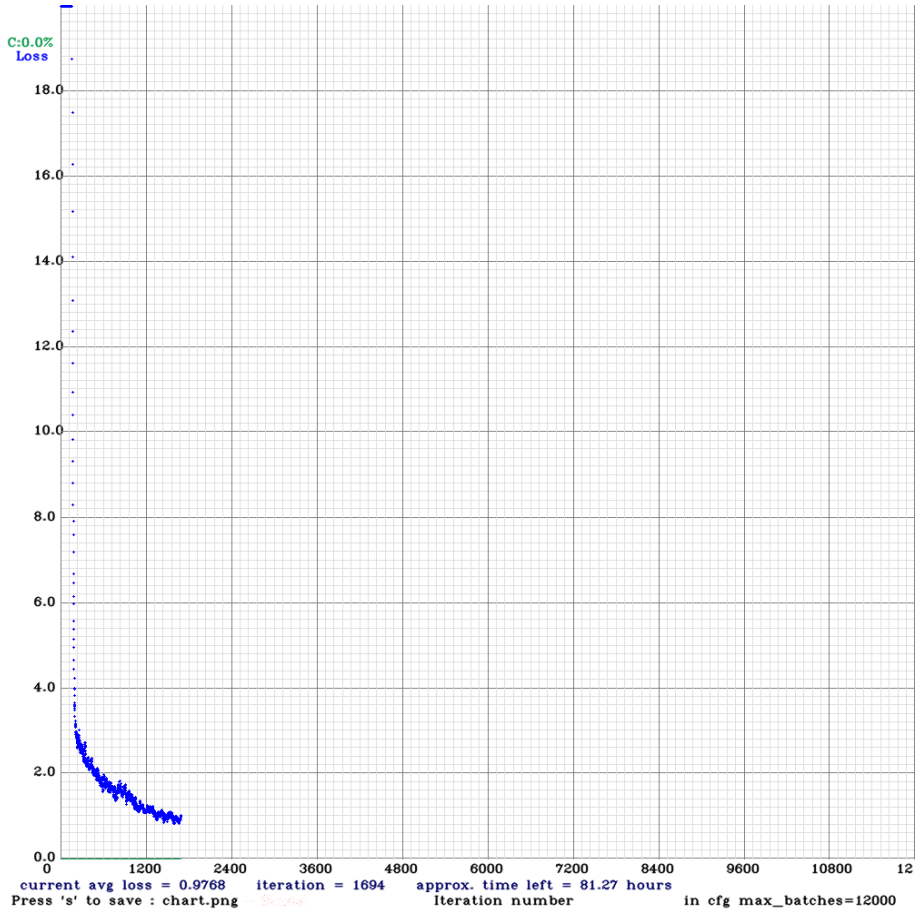


Figura 5.1 Gráfica de la evolución de la *loss* respecto a las iteraciones a lo largo del entrenamiento de YOLO.

Al ver que la *loss* seguía prácticamente estable en un valor tan alto surgieron sospechas de que algo no iba bien, y es cuando se pasó a comprobar las imágenes del *dataset*, averiguando que ciertas *bounding boxes* no habían sido capturadas de forma válida, por lo que había muchas imágenes que no estaban etiquetadas. Durante un prolongado periodo de marcado, las 8.210 imágenes del *dataset* fueron de nuevo clasificadas correctamente de forma manual para, esta vez sí, realizar un entrenamiento preciso con ellas.

Este entrenamiento partió de la base del anterior, utilizando sus pesos en lugar de los pesos de YOLOv4 como partida de inicio, lo que haría que el proceso fuera más rápido ya que estos sí "conocían" las cajas, sólo que se estaba usando información ambivalente con las imágenes mal etiquetadas, lo que provocaba que la evolución se estancara. El segundo entrenamiento, como se puede apreciar en la Figura 5.2, se desarrolló con más celeridad, viéndose una tendencia decreciente del error a medida que aumentaba el número de iteraciones.

El proceso fue detenido en la iteración 10.526, de las cuales aproximadamente 6.800 fueron realizadas con el *dataset* en estado defectuoso y alrededor de 3.700 iteraciones una vez corregido. El periodo total de entrenamiento tomó cerca de 60 horas, siendo alrededor de 38 horas el primer tramo y en torno a las 22 horas el segundo. Una vez obtenidos los pesos definitivos que iban a ser usados para la detección, fueron trasladados a la Jetson Nano, en la cual se realizaron diferentes experimentos. En primer lugar se ejecutó el proceso de validación mediante el comando que se presenta en el repositorio de GitHub, sólo que en lugar de aplicarse sobre las imágenes del *dataset* COCO, se ejecutará sobre las propias del proyecto.

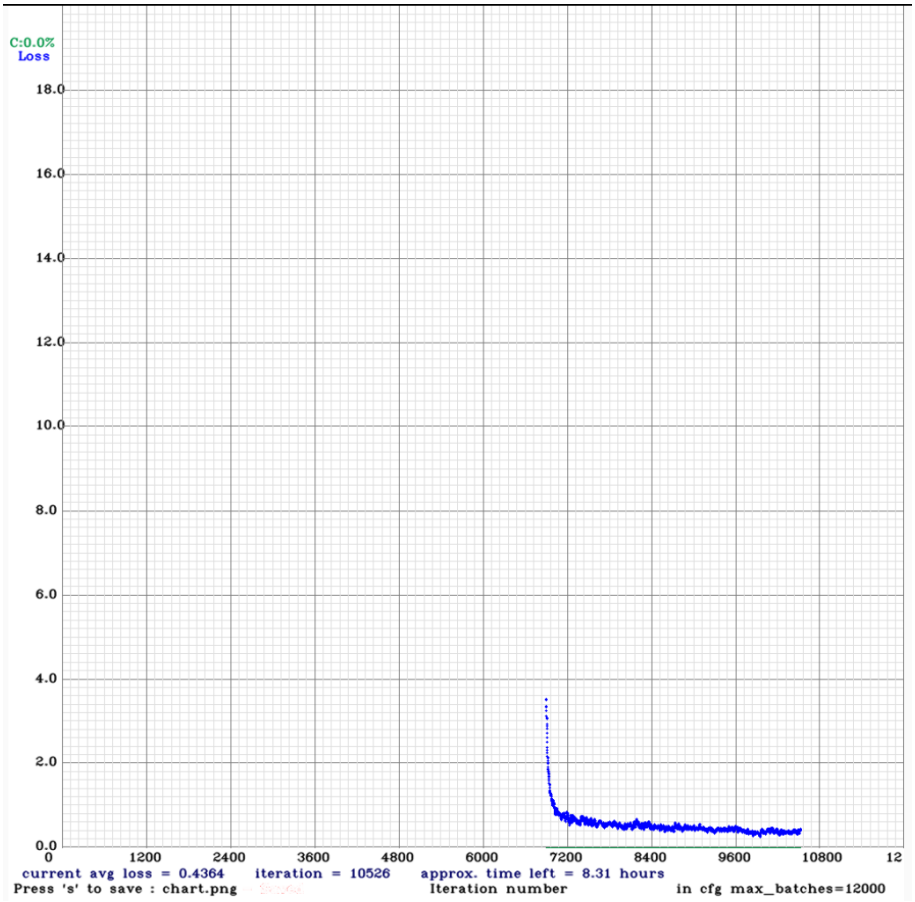


Figura 5.2 Gráfica de la evolución de la *loss* respecto a las iteraciones a lo largo del segundo entrenamiento de YOLO.

```
$ ./darknet detector valid build/darknet/x64/data/tfg.data cfg/yolo-tfg.cfg backup
    /yolo-tfg_last.weights
```

Esta línea de comando proporciona seis archivos en el directorio `/darknet/results`, uno por cada tipo de clase del *dataset*. El interior de estos archivos está dividido en seis columnas: la primera representa en qué imagen de validación ha sido detectado el objeto que da nombre al fichero, la segunda indica la *confidence score* o nivel de confianza, es decir, lo segura que está la red de que su predicción es correcta, la tercera y cuarta indican la distancia de la esquina superior izquierda de la *bounding box* respecto al límite izquierdo y el borde superior respectivamente de la imagen, mientras que las columnas quinta y sexta marcan las distancia de la esquina inferior derecha del recuadro también con respecto al límite izquierdo y el borde superior respectivamente de la fotografía. Por tanto, sigue otros criterios que los que usaba el programa *Yolo_mark*, en los que se indicaba la posición con respecto **al centro** de la *bounding box*, y la siguientes columnas indicaban el ancho y el alto, de ahí que se verá a continuación cómo solucionarlo.

Cuando hay más de una línea con el mismo nombre de imagen en la primera columna significa que se han realizado tantos *bounding boxes* en la imagen como líneas haya, pero sólo se tomarán como válidos aquellos que tengan una precisión mayor o igual que .25. Este valor es conocido como *threshold*, y puede ser modificado mediante una simple *flag* `-thresh <val>` junto al comando de detección. Por ejemplo, si se desea que se muestren en la imagen todas las *bounding boxes* detectadas independientemente de su *nivel de confianza*, el *threshold* puede ser establecido a cero

como en el siguiente ejemplo:

```
$ ./darknet detect cfg/yolov4.cfg yolov4.weights data/dog.jpg -thresh 0
```

Muestra también de los ficheros de texto con los resultados descritos serían las siguientes líneas pertenecientes al archivo que representa las predicciones sobre la caja azul, el cual es guardado con el nombre `comp4_det_test_azul.txt`:

```
valid_00001 0.002431 106.677048 197.819427 233.440811 262.366638
valid_00001 0.993573 375.050476 41.978973 499.943237 262.090485
valid_00001 0.012134 358.094116 36.840240 539.543091 289.311981
valid_00001 0.002370 390.011139 71.109070 485.608612 229.140869
valid_00002 0.002098 109.268311 197.356354 230.849487 262.444397
valid_00002 0.001061 82.666801 261.013611 236.525909 341.182251
valid_00002 0.990567 367.791687 37.304558 503.936401 268.819000
valid_00002 0.002101 389.807831 70.046844 485.204315 231.305206
```

En estas líneas se aprecia cómo en la primera imagen destinada a validación, `valid_00001.jpg`, han sido detectadas cuatro posibles cajas azules, pero sólo la de la segunda línea es la que será representada, ya que tiene un *confidence score* mayor o igual que .25, de hecho su precisión en este caso es muy elevada, 0.993573. Centrando la vista en esta *bounding box*, su esquina superior izquierda se encuentra a 375 píxeles del borde izquierdo de la imagen, y a 42 píxeles del borde superior, como se justifica en la Figura 5.3. Para calcular el ancho y el alto habría que restar la tercera columna con la quinta, y la cuarta con la sexta, respectivamente. En la segunda imagen de validación, `valid_00002.jpg`, vuelve a ocurrir la misma situación, cuatro *bounding boxes* son detectadas pero sólo una de ellas será tomada en cuenta, en este caso la tercera.

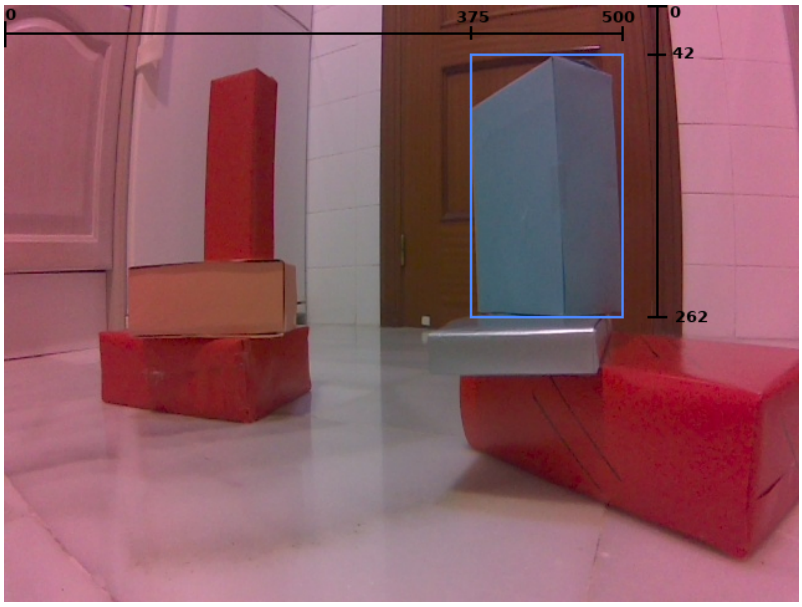


Figura 5.3 Explicación visual del formato usado por YOLO para representar la *bounding box*.

Para poder trabajar con estos datos se realizaron una serie de procedimientos y modificaciones con la intención de comparar con los datos reales. En primer lugar se eliminaron las líneas de los archivos con una nivel de confianza menor que 0.25, ya que los resultados de YOLO tampoco los

mostrarían, como ya se ha mencionado. Tras ello, se añadió una nueva columna con el ID del objeto identificado para poder unificar todos las predicciones en un mismo archivo pero seguir pudiendo ser capaz de individualizar e identificar las predicciones asociadas con cada clase.

Con el fin de ordenar los datos de forma más visual se usará una hoja de cálculo, en la cual se han dispuesto las predicciones y los valores reales de forma que estén ordenados por el número de la imagen a la que representan. Para poder comparar estos valores primero es necesario pasarlo al mismo formato, ya que los resultados de validación calculados por YOLO están directamente en píxeles, mientras que los realizados con el programa *Yolo_mark* son relativos al ancho y alto de la imagen. Además, los primeros representan las distancias desde la esquina superior izquierda e inferior derecha hasta el borde de la imagen, como se ha explicado anteriormente, mientras que los segundos son expresados con respecto al centro de la *bounding box*, dando posteriormente el valor de su ancho y alto. Por lo tanto, se convertirán todos los de *Yolo_mark* al formato de los resultados de validación de YOLO, ya que se considera más práctico para los procedimientos a realizar.

Será necesario multiplicar los valores referentes al eje *x* por 640, que es el ancho de las imágenes, y los referentes al eje *y* por 480, que es la altura. Con esto se obtendrá los valores en píxeles de posición del recuadro, y su ancho y alto, por lo que se sumará y restará la mitad de estos a la posición para obtener la posición de la esquina superior izquierda y la inferior derecha, obteniendo así los resultados tal como se expresan con comando `./darknet detector valid`.

Para comparar los datos se calculará el *Intersection over Union* (IoU) de cada una de las predicciones realizadas por YOLO, tal como se vio en la Figura 3.15. En primer lugar se realizó el cálculo de la IoU para todas las predicciones con nivel de confianza mayor que 0.25, incluidas aquellas que eran consideradas falsos positivos, lo cual daba resultados muy dispares, los cuales pueden verse en las gráficas siguientes.

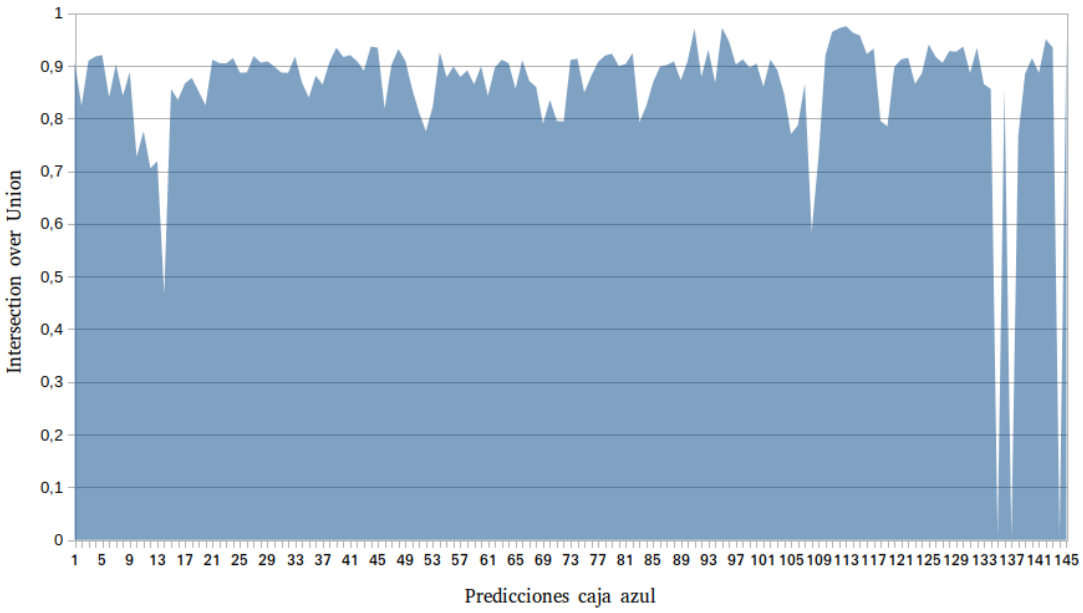


Figura 5.4 Intersection over Union de predicciones caja azul detectados por YOLO.

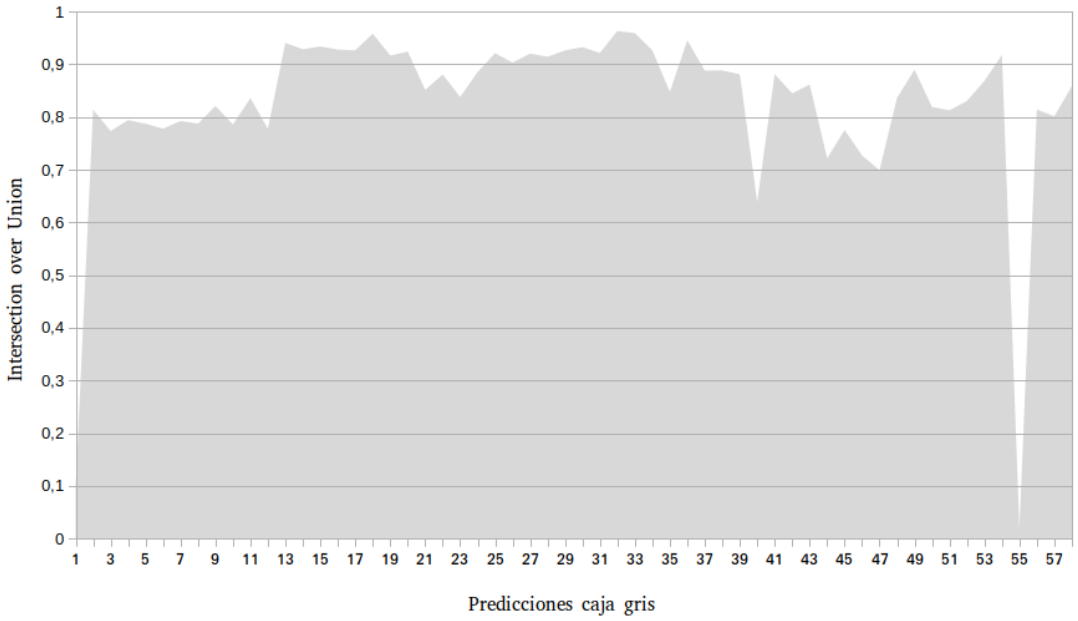


Figura 5.5 Intersection over Union de predicciones caja gris detectados por YOLO.

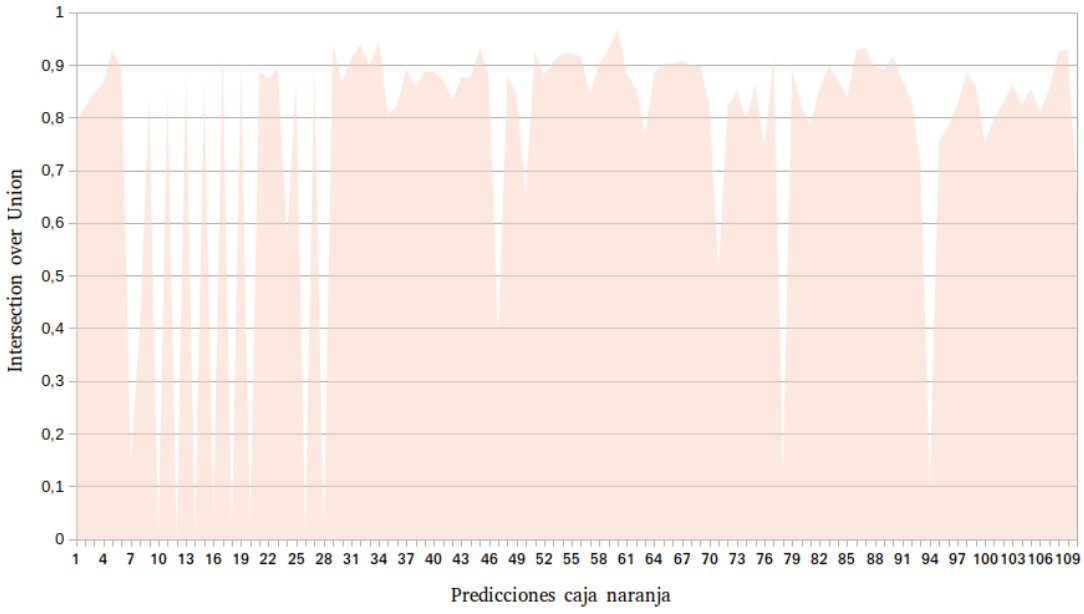


Figura 5.6 Intersection over Union de predicciones caja naranja detectados por YOLO.

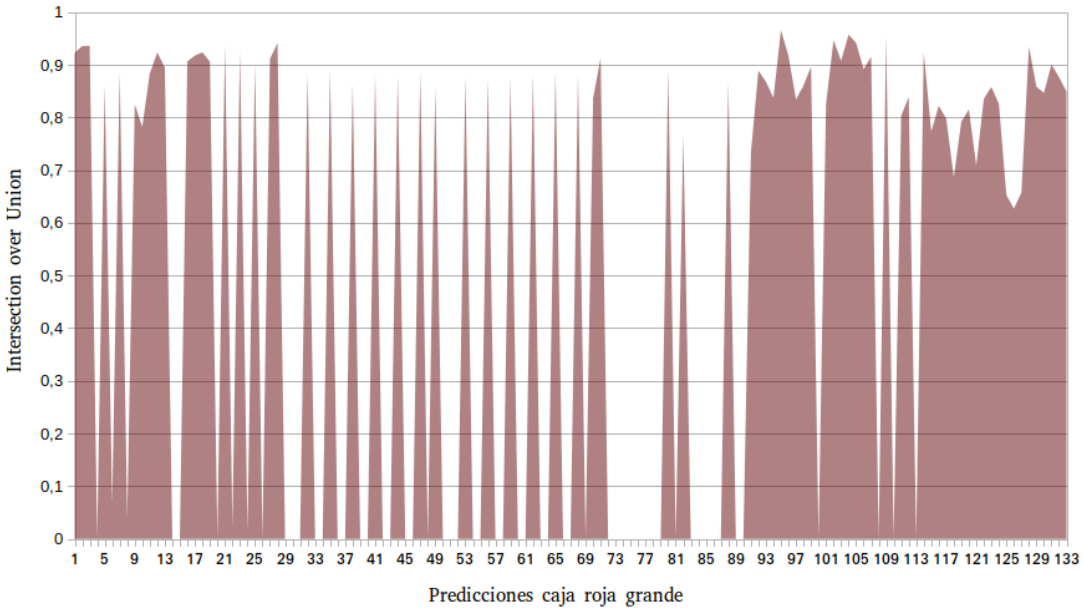


Figura 5.7 Intersection over Union de predicciones caja roja grande detectados por YOLO.

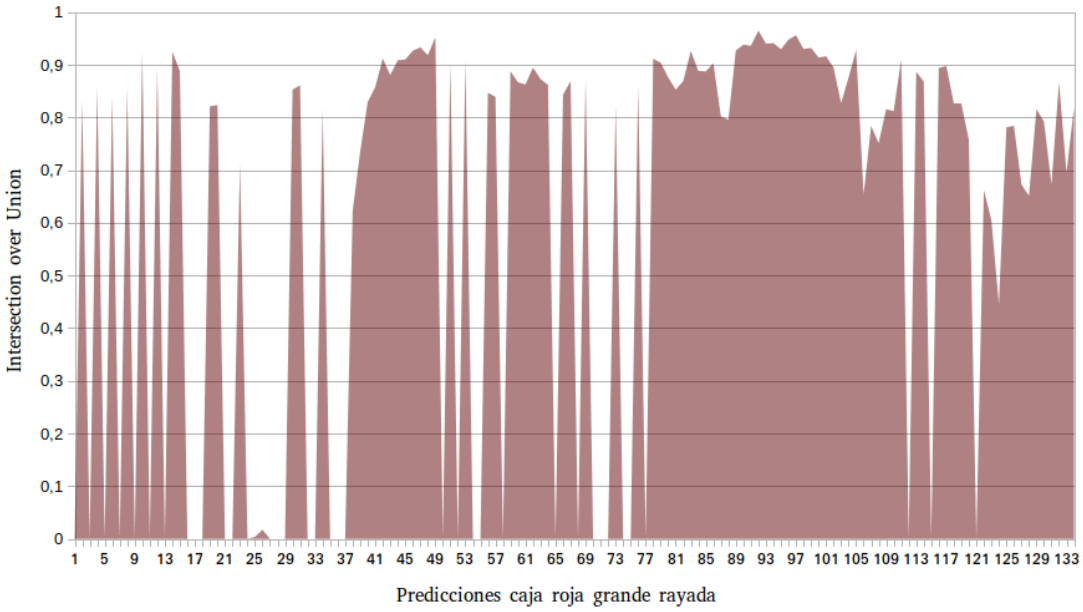


Figura 5.8 Intersection over Union de predicciones caja roja grande rayada detectados por YOLO.

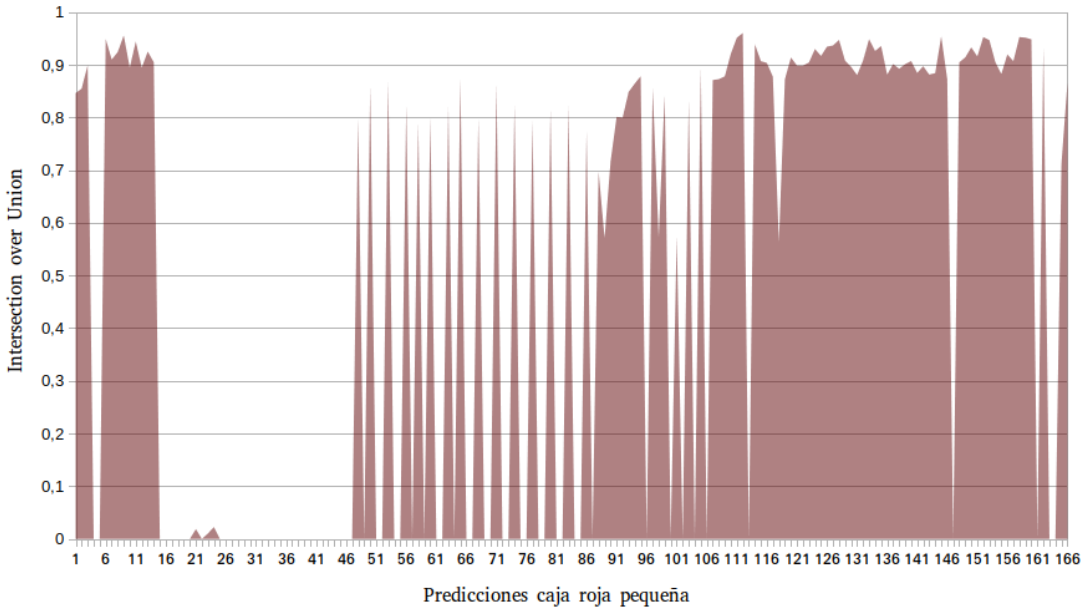


Figura 5.9 Intersection over Union de predicciones caja roja pequeña detectados por YOLO.

Tabla 5.1 IoU promedio teniendo en cuenta todas las predicciones con nivel de confianza mayor o igual que 0.25.

Clase	IoU promedio	Predicciones
Azul	0.8599	145
Gris	0.8294	59
Naranja	0.7684	111
RojoBig	0.4969	134
RojoBigrayado	0.6023	135
RojoSmall	0.5052	167
Total	0.6540	751

Viendo estos resultados apreciables en las gráficas se determinó que el criterio empleado no estaba siendo justo con el modelo, ya que se estaba calculando la IoU también en las predicciones falladas, por lo que esto alejaba el sentido del IoU, que es determinar cómo de bien la red traza el *bounding box* cuando acierta el objeto. Es por ello que se volvió a repetir las gráficas pero filtrando por casos verdaderos positivos con un *threshold* de 0.3, lo cual da los resultados mostrados a continuación.

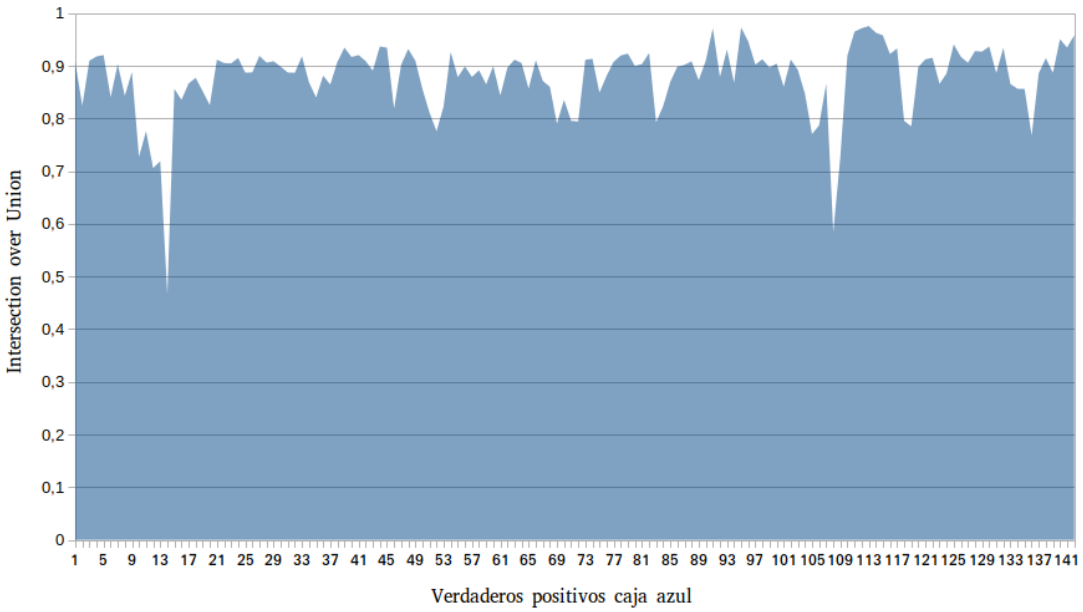


Figura 5.10 Intersection over Union de verdaderos positivos caja azul detectados por YOLO.

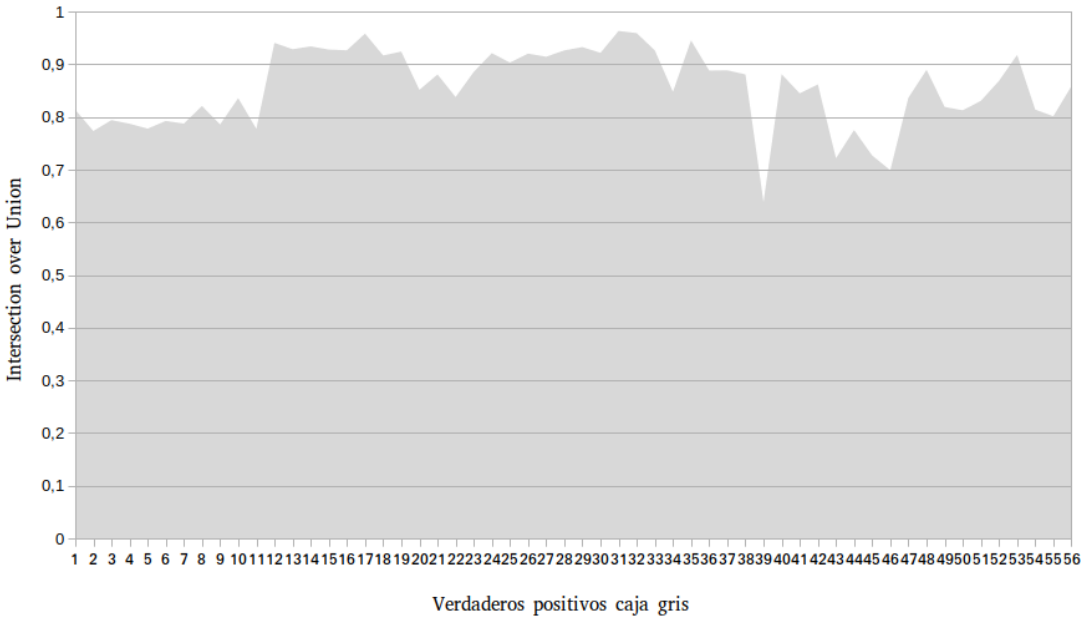


Figura 5.11 Intersection over Union de verdaderos positivos caja gris detectados por YOLO.

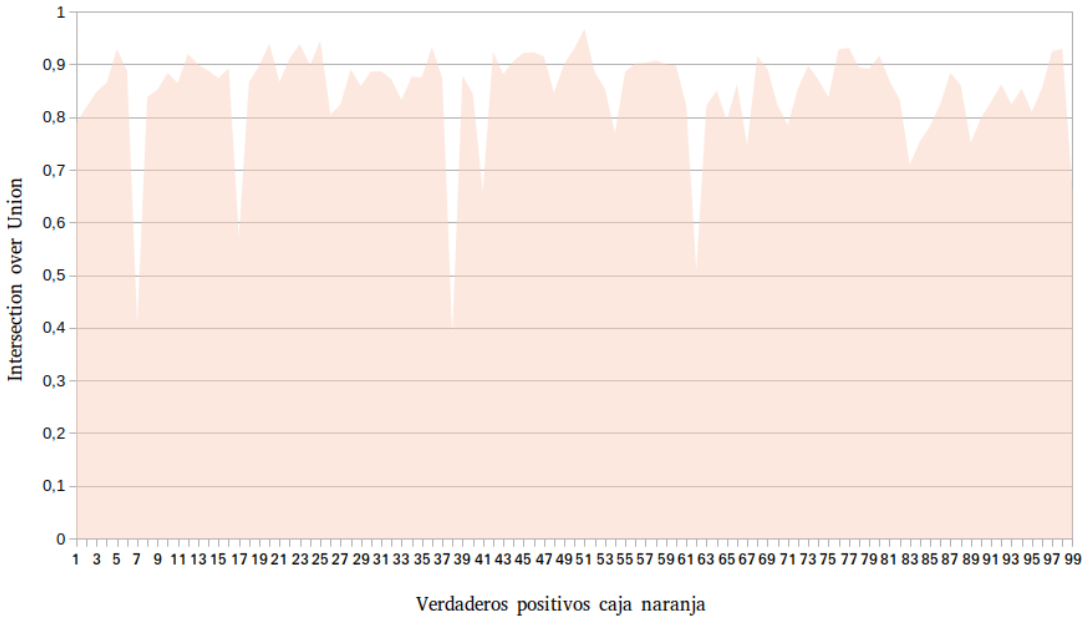


Figura 5.12 Intersection over Union de verdaderos positivos caja naranja detectados por YOLO.

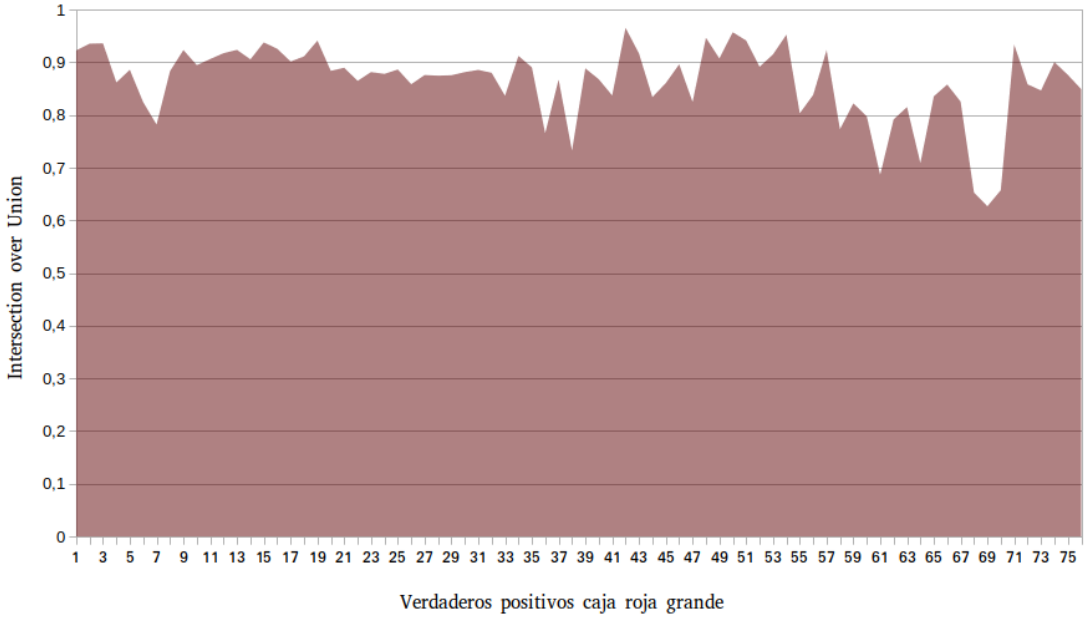


Figura 5.13 Intersection over Union de verdaderos positivos caja roja grande detectados por YOLO.

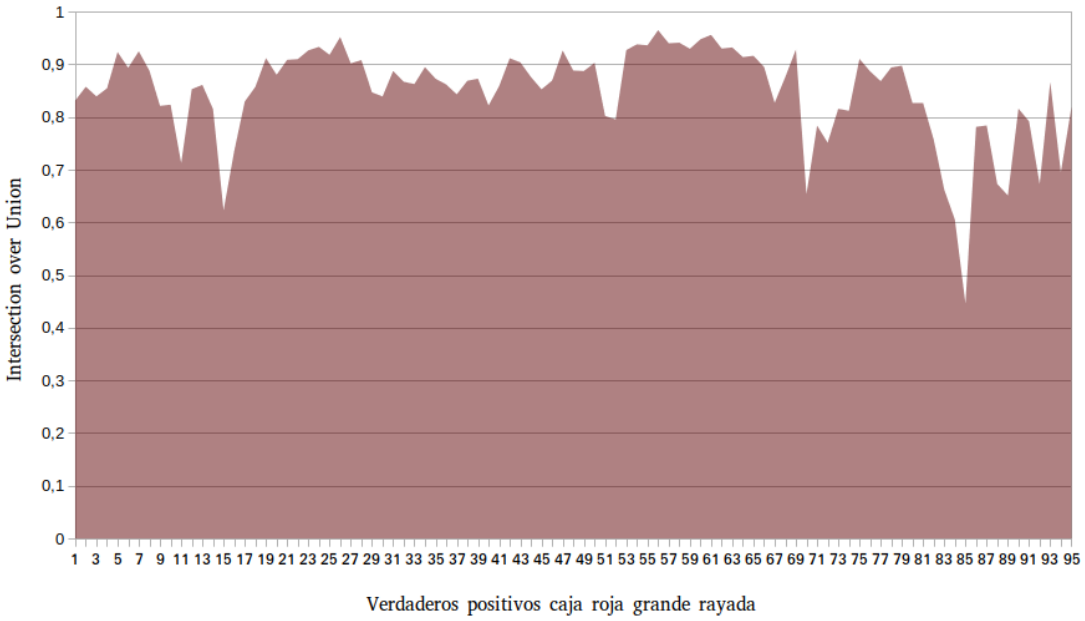


Figura 5.14 Intersection over Union de verdaderos positivos caja roja grande rayada detectados por YOLO.

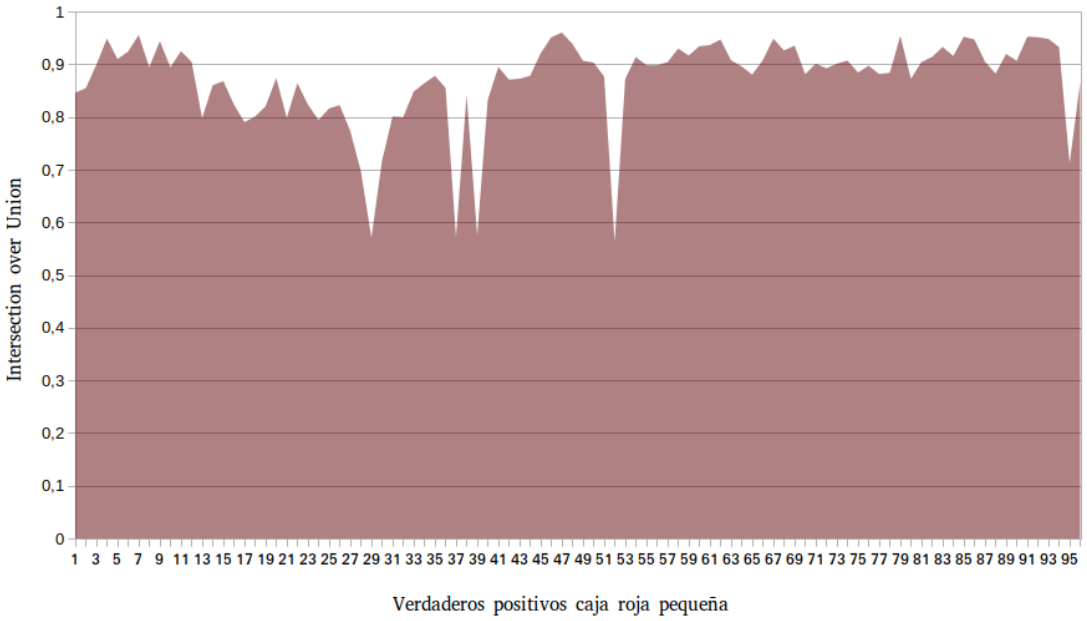


Figura 5.15 Intersection over Union de verdaderos positivos caja roja pequeña detectados por YOLO.

Estos resultados ofrecen una información mucho más clara y honesta de las características de la

Tabla 5.2 IoU promedio *true positives* con *threshold* de 0.3.

Clase	IoU promedio	True Positives@0.3
Azul	0.8778	142
Gris	0.8564	56
Naranja	0.8484	100
RojoBig	0.8630	77
RojoBigrayado	0.8466	96
RojoSmall	0.8693	97
Total	0.8619	568

red que las gráficas expuestas anteriormente, indicando que, cuando el objeto es detectado de forma correcta, la realización de la *bounding box* es, en general, bastante correcta. De hecho, los picos más bajos se deben a situaciones en las que la caja no es visible completamente. Por ejemplo, el valor más bajo que se observa en la Figura 5.10 es 0,46514, el cual se da en la imagen de validación `valid_00014.jpg`. Si se procede a analizarla con YOLO se obtiene el resultado de la Figura 5.16, en el cual puede comprobarse que este bajo valor se debe a que casi la mitad de la caja azul se encuentra tapada por la caja naranja, lo que hace que no se detecte de forma correcta, pero se trata de algo razonable ya que el caso es complicado.

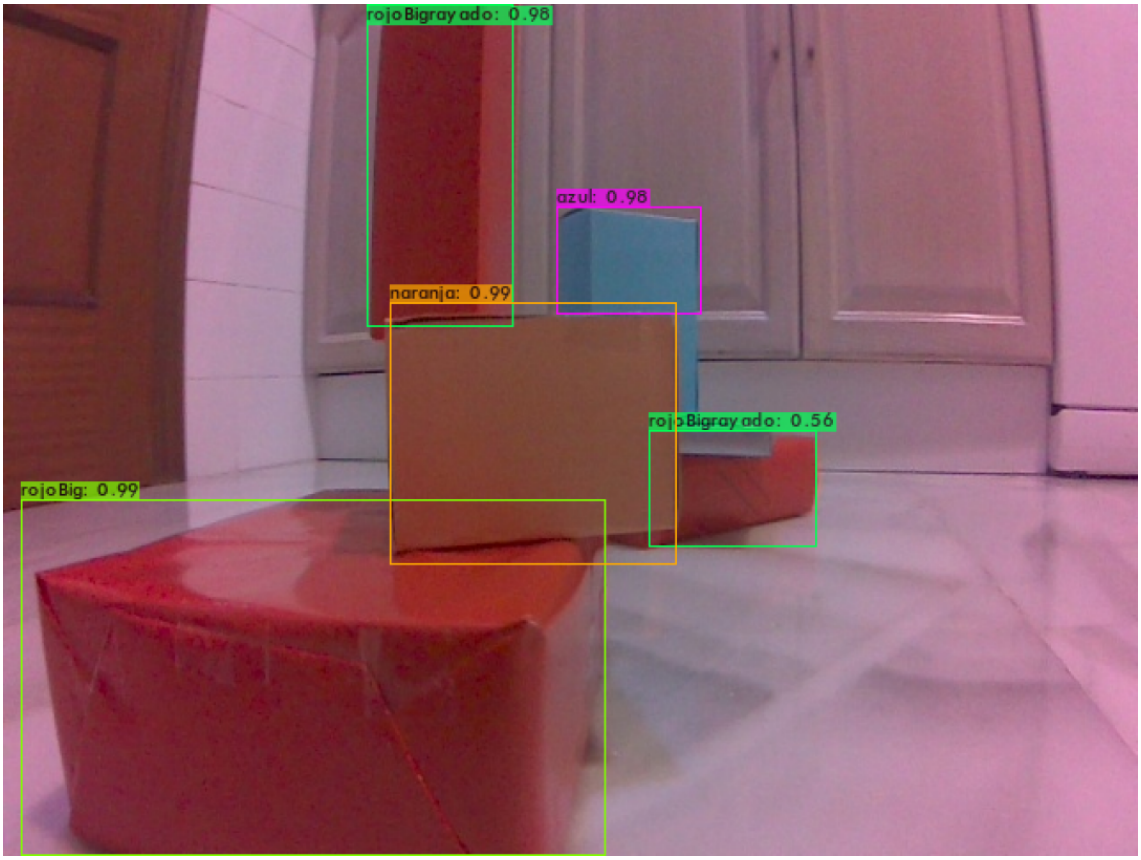


Figura 5.16 Salida de YOLO para la imagen con IoU más bajo de los verdaderos positivos de la clase azul.

En esta misma imagen también se puede apreciar cómo la caja gris se encuentra tan tapada por otras y se ve tan pequeña que la red no es capaz de detectarla, siendo algo característico de YOLO su limitación para detectar objetos pequeños. La caja roja pequeña, por su parte, se encuentra encima de la caja naranja, causando que tampoco sea clasificada de forma correcta y sea confundida con la clase grande rayada. Esto es algo comprensible ya que las imágenes del *dataset* en las que la caja roja rayada se veía a lo lejos, la cámara no era capaz de captar las rayas de la misma, por lo que ocasionó que el modelo fuera desarrollado con imágenes que podían dar lugar a confusión. Pese a ello, se considera que es una complejidad coherente para este tipo de redes, ya que en un funcionamiento real no siempre se van a tener las condiciones de visión idóneas para apreciar todos los detalles de los objetos con claridad. La media del *Intersection over Union* de todos los verdaderos positivos juntando todas las clases es de 0.8618, un valor bastante elevado indicando que el punto débil de la red no se encuentra en el trazado del recuadro que delimite al objeto.

Para medir cómo de bien el modelo detecta todos los positivos se calculará el ***recall*** mediante la fórmula contenida en la Figura 5.17, para lo cual hará falta calcular todos los verdaderos positivos y falsos negativos, algo fácil con la base de datos que se ha creado. Con el fin de evaluarlo, se considerará un IoU *threshold* del 50 % para determinar si un resultado es verdadero positivo o no. De esta forma se obtiene la siguiente Tabla 5.3:

Tabla 5.3 *Intersection over Union*, *True Positives*, *False Positives*, *False Negatives*, casos totales, *recall* y precisión, con un *threshold* de 0.5.

Clase	IoU promedio	TP@0.5	FP@0.5	FN@0.5	Apariciones reales	Recall	Precisión
Azul	0.8778	140	4	8	148	0.9459	0.9722
Gris	0.8564	57	2	46	103	0.5534	0.9661
Naranja	0.8484	97	13	28	125	0.7760	0.8818
RojoBig	0.8630	76	57	33	109	0.6972	0.5714
RojoBigrayado	0.8466	93	39	29	122	0.7623	0.7045
RojoSmall	0.8693	94	46	26	120	0.7833	0.6714
Total	0.8649	564	161	163	727	0.7758	0.7779

Para calcular los valores de *recall* y *precision* se han tenido en cuenta las dos fórmulas siguientes:

$$Recall = \frac{TP}{TP + FN}$$

Figura 5.17 Fórmula para calcular el *recall* [66].

$$Precision = \frac{TP}{TP + FP}$$

Figura 5.18 Fórmula para calcular la precisión.

En base a ellas puede señalarse que el *recall* es la proporción de verdaderos positivos en relación a todos los posibles positivos, es decir, en relación a todas las apariciones reales. Por su parte, la precisión mide cómo de correctas son las predicciones, es decir, el porcentaje de predicciones correctas. Cabe aclarar que *precision* no es lo mismo que *average precision* (AP), ya que la primera es calculada como se observa en la Figura 5.18, mientras que la segunda corresponde al área bajo la curva que se obtiene al representar *precision* frente a *recall*, por lo que se calcularía mediante su integral.

Por otra parte, la *mean Average Precision* (mAP) a veces es confundida con la AP, incluso en muchos contextos como en el dataset de COCO son usadas indistintamente. Formalmente, mAP es la media de diferentes AP calculadas sobre distintos valores de *threshold* para la IoU, pero en otros contextos, en contraste, se calculan la AP para cada clase en particular y se realiza la media de ellas, usando un sólo valor de *threshold*. Este último procedimiento es el que realiza YOLO con otro importante comando que aporta información sobre el *dataset* destinado a validación, el cual se muestra a continuación:

```
./darknet detector map build/darknet/x64/data/tfg.data cfg/yolo-tfg.cfg backup/
    yolo-tfg_last.weights
```

Lo cual devuelve por pantalla la información observable en la Figura 5.19.

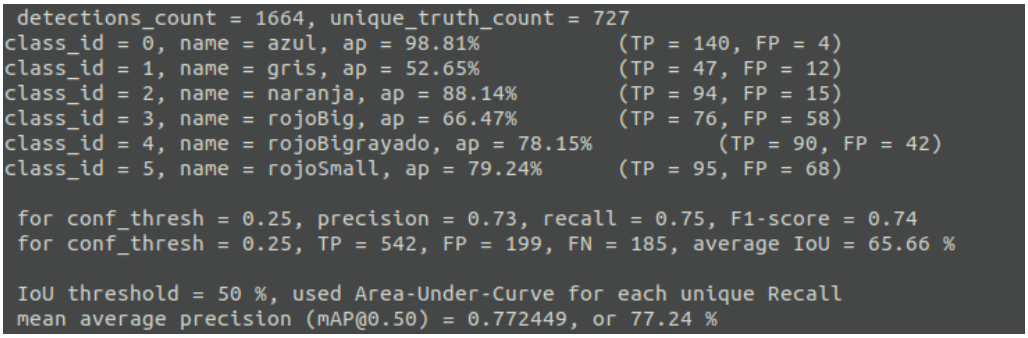


Figura 5.19 Respuesta de la terminal al comando de *mean average precision* (mAP) sobre el *dataset* de validación.

Cabe destacar que los valores de verdaderos positivos, falsos positivos y falsos negativos extraídos por este comando son ligeramente distintos a los que fueron determinados manualmente en la Tabla 5.3, lo cual puede deberse a ciertas diferencias entre la metodología escogida por YOLO para su consideración y la metodología que se ha usado particularmente en los cálculos que se realizaron para este proyecto.

Los resultados obtenidos muestran una serie de parámetros del comportamiento de la red sobre los datos de validación, entre los que se encuentran el porcentaje de precisión media para cada una de las clases, así como su número de verdaderos positivos y falsos positivos. Esta información ha sido representada en la Tabla 5.4 para una mejor visualización.

En la tabla puede apreciarse que el objeto con mejor precisión media, 52.65 %, es la caja gris, la cual coincide con que es la más pequeña de todas ellas. La deficiencia de YOLO con la detección de objetos pequeños es una problemática que caracteriza a la propia red, lo cual fue ya mencionado en el apartado 3.5, por lo que es lógico que se dé este resultado justamente en la caja de menor tamaño.

Por otra parte, la clase con mayor precisión media corresponde con la caja azul, tratándose del elevado porcentaje de 98.81 %. Este objeto es el más fácil de identificar, debido a su diferencial

Tabla 5.4 Esquematización de la fiabilidad de la red con cada objeto.

ID	Caja	Average Precision	True Positive	False Positive
0	Azul	98.81 %	140	4
1	Gris	52.65 %	47	12
2	Naranja	88.14 %	94	15
3	RojoBig	66.47 %	76	58
4	RojoBigrayado	78.15 %	90	42
5	RojoSmall	79.24 %	95	68
-	Total	mAP@0.50=77.24 %	542	199

color respecto a las otras cajas que presentan colores rojizos y anaranjados, fácilmente confundibles entre ellos, circunstancia que se agravaba con la baja calidad de la cámara, la cual aplicaba tonos demasiado rojizos en ciertas ocasiones a las imágenes como si de un filtro se tratase. Además, el tamaño de la caja es bastante más voluminoso que la caja gris, por lo cual no se da el fenómeno de dificultad en el reconocimiento de objetos pequeños que sí se daba en la caja grisácea. Con toda esta información es razonable que se trate de la clase mejor identificada de todas las presentes, y el tener un color tan distintivo hace que el número de falsos positivos se reduzca únicamente a 4, una inclinación enorme frente a 140 verdaderos positivos.

Las cuatro cajas restantes son aquellas que son fácilmente confundibles entre ellas, principalmente las tres rojas entre sí. Como es natural, la caja naranja es la que más precisión media presenta de ellas, un 88.14 %, y colocándose en el segundo lugar de la clasificación. Pese a haber situaciones en las que el color representado en las imágenes puede ser dudoso, la red ha sido capaz de salvar con mucha holgura su cometido en la identificación de esta clase.

Centrándose en los resultados obtenidos de las tres cajas rojas, dos de ellas tienen una precisión media muy pareja y más elevada que la caja roja grande sin rayas. Este resultado es también esperado, ya que, en primer lugar, las rayas en las paredes del objeto son fácilmente distinguibles respecto las otras dos cajas lisas en situaciones de buena visibilidad, pero teniendo en cuenta la baja resolución de las imágenes y el poco detalle del sensor de la cámara, esto sólo suele suceder en imágenes en las que la caja está más cerca y las rayas del objeto puedan verse representadas en la fotografía. En segundo lugar, la caja pequeña puede ser confundida con la caja grande en muchas ocasiones, pero la diferente relación de aspecto de sus dimensiones con respecto a las de la caja grande puede haber sido un factor determinante a la hora de su identificación, por lo que se cree que este es uno de los motivos por los que obtiene un porcentaje de acierto considerablemente mayor que la caja grande lisa.

La caja con peor resultado de todas ellas, por tanto, es la grande sin rayar, con una precisión media del 66.47 %. Se trata del segundo peor resultado por detrás de la identificación de la caja gris, y también tiene el ratio más igualado entre verdaderos positivos y falsos positivos. Algo alarmante de las tres cajas rojas es que la cantidad de falsos positivos se dispara en comparación con las otras tres cajas. Esto se debe principalmente a lo fácil que es confundirlas entre ellas, reto por el cual fueron introducidas en la realización de este proyecto. Peso a ello, los resultados obtenidos son favorables y útiles en gran parte de los casos. La precisión obtenida con este comando es 0.73 y el *recall* 0.75, indicando que el modelo detecta los verdaderos positivos en un porcentaje bastante elevado, y de sus predicciones totales son correctas también una amplia mayoría, aunque estos valores son ligeramente inferiores a los que fueron calculados manualmente. Para finalizar el inciso en el que se ha estado comentando los datos de validación, se incluirán varias fotografías que demuestran la naturaleza de la red, tanto con fallos como con aciertos:

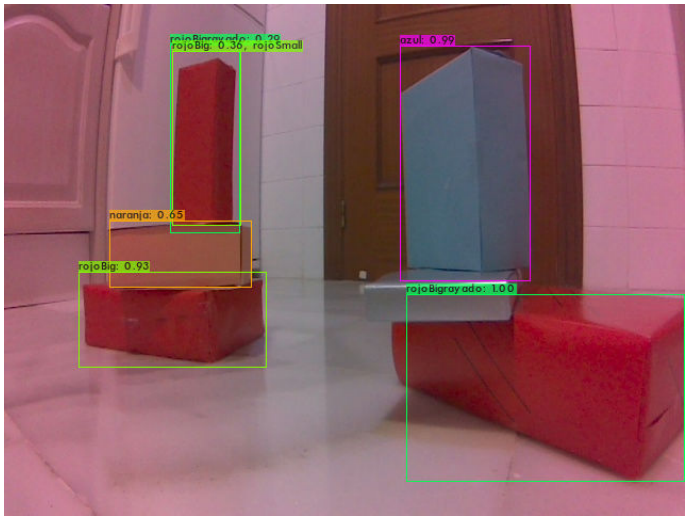


Figura 5.20 Predicción de la imagen *valid_00004.png*.

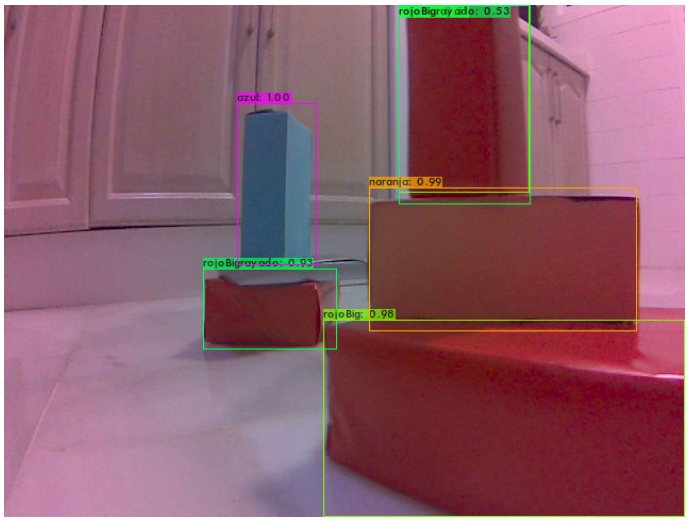


Figura 5.21 Predicción de la imagen *valid_00019.png*.

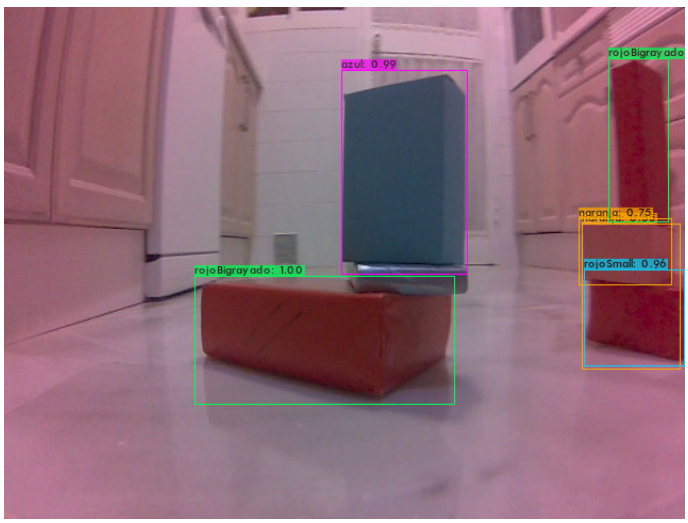


Figura 5.22 Predicción de la imagen *valid_00047.png*.

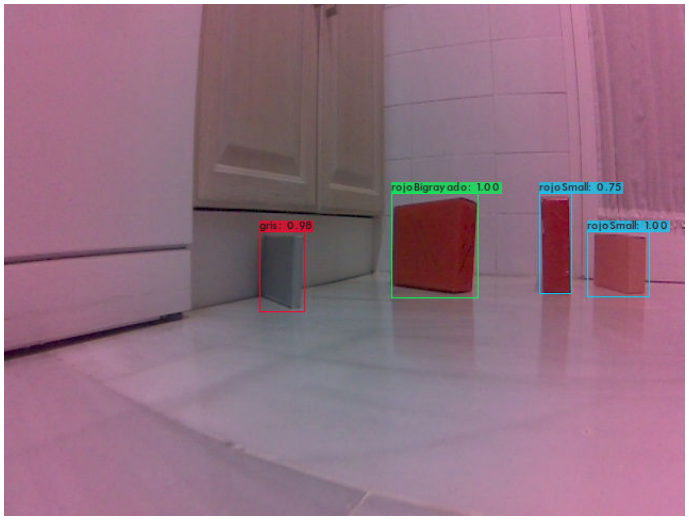


Figura 5.23 Predicción de la imagen *valid_00054.png*.

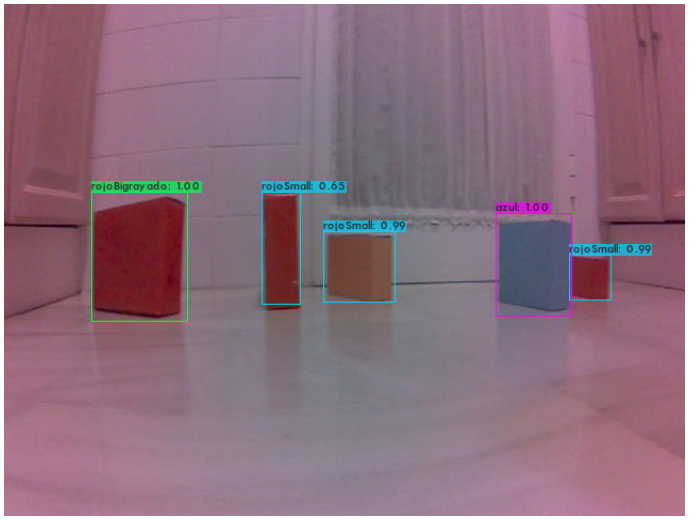


Figura 5.24 Predicción de la imagen *valid_00060.png*.

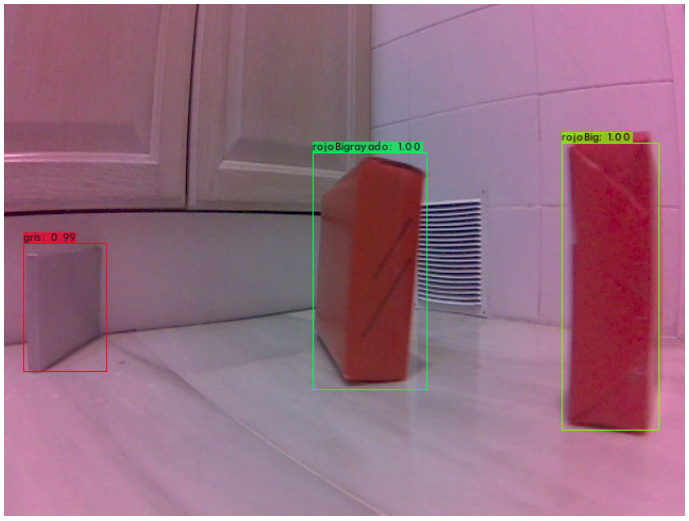


Figura 5.25 Predicción de la imagen *valid_00107.png*.

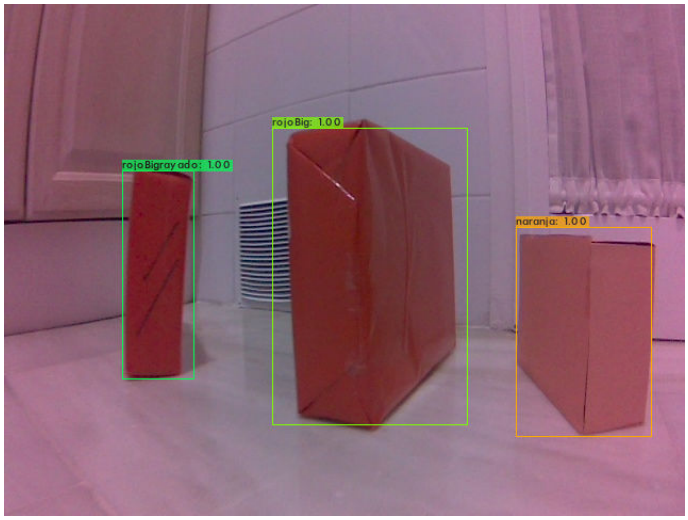


Figura 5.26 Predicción de la imagen *valid_00111.png*.

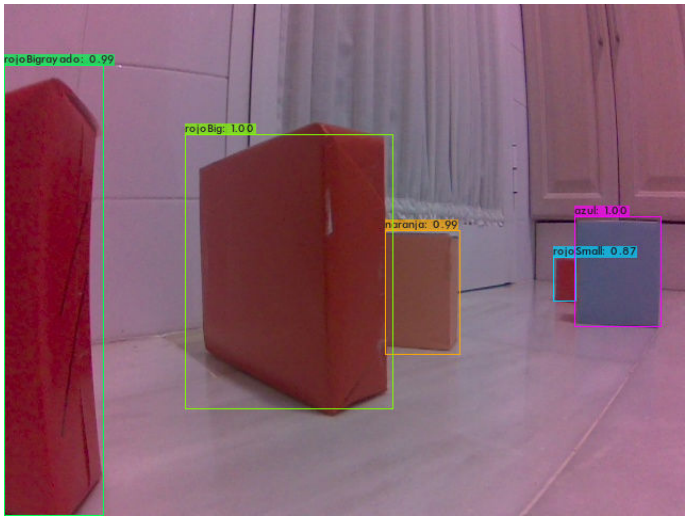


Figura 5.27 Predicción de la imagen *valid_00123.png*.

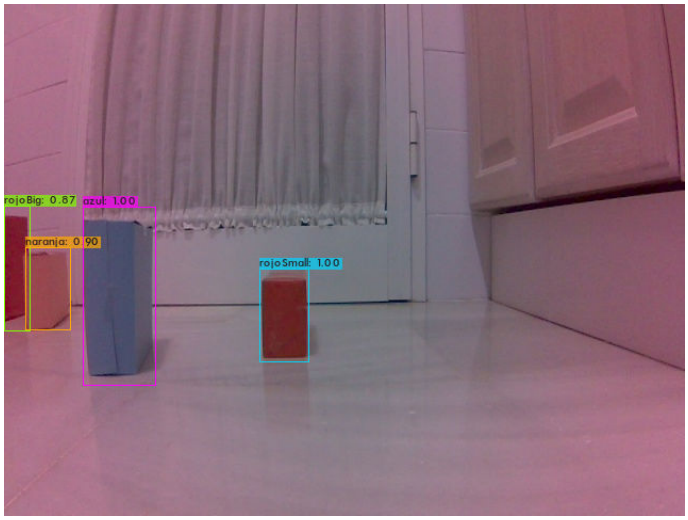


Figura 5.28 Predicción de la imagen *valid_00130.png*.

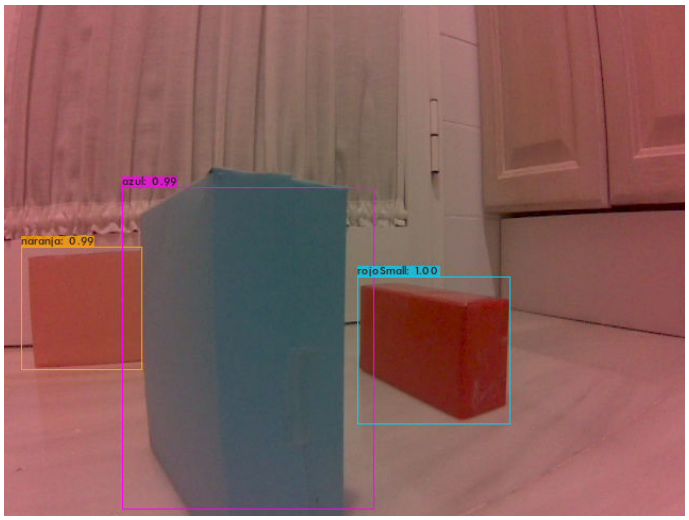


Figura 5.29 Predicción de la imagen *valid_00170.png*.

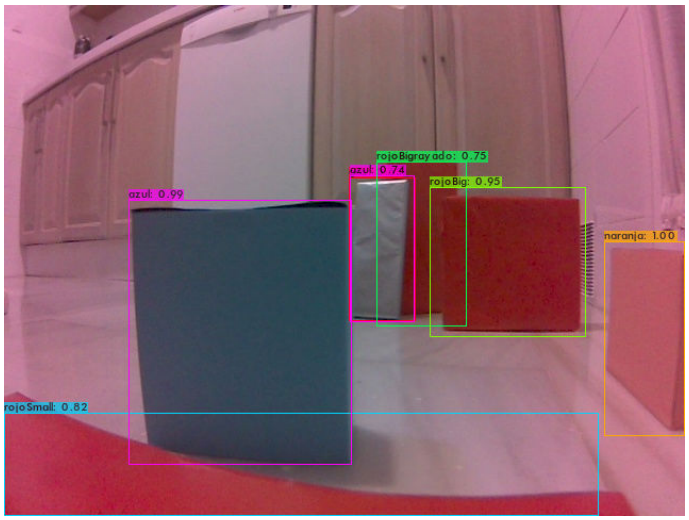


Figura 5.30 Predicción de la imagen *valid_00200.png*.

Para la prueba del modelo en experimentos reales, sin pruebas de validación, se han realizado una serie de vídeos que realizan las *bounding boxes* en vídeos grabados previamente y luego pasados por la red neuronal. Con el fin de caracterizar su comportamiento en tiempo real también se han realizado grabaciones de pantalla de la Jetson Nano, mostrando con ello la tasa de frames en directo. Estos vídeos han sido subidos a la plataforma YouTube, y pueden ser accedidos mediante los siguientes *hyperlinks*:

- *Test* en tiempo real de la pantalla de la Jetson Nano
- *Test* 1 - Video grabado y luego procesado por YOLO
- *Test* 2 - Video grabado y luego procesado por YOLO
- *Test* 3 - Video grabado y luego procesado por YOLO
- *Test* 4 - Video grabado y luego procesado por YOLO
- *Test* 5 - Video grabado y luego procesado por YOLO

En ellos se pueden ver cómo la red neuronal se ejecuta durante un gran número de minutos, en los que se puede apreciar su funcionamiento con fallos y errores. El primer vídeo corresponde con la ejecución de YOLO en tiempo real corriendo en la Jetson Nano, durante una larga prueba de alrededor de 9 minutos. La ejecución de YOLO es iniciada con el comando `./darknet detector demo`, seguida de la ubicación de los archivos `.data`, `.cfg` y el fichero de pesos `.weights`. En el repositorio oficial indica que para usar la cámara hay que usar la etiqueta `-c 0`, pero esto no funcionaba en la Jetson, lo cual fue solucionado introduciendo una serie de parámetros descriptivos de la cámara, así como la resolución escogida de 640×480 ya que así fue entrenada la red. Esto se encuentra recogido todo en el siguiente comando:

```
./darknet detector demo tfg/tfg.data tfg/yolo-tfg.cfg tfg/tfg.weights "
    nvarguscamerasrc ! video/x-raw(memory:NVMM), width=640, height=480, format=(
    string)NV12, framerate=(fraction)30/1 ! nvvidconv flip-method=0 ! video/x-raw,
     format=(string)BGRx ! videoconvert ! video/x-raw, format=(string)BGR !
    appsink"
```

Durante la ejecución del programa el robot fue movido por el espacio de trabajo, cambiando las localizaciones de las cajas, apilándolas, girándolas, etc. Con esto se comprobó que se obtenía un resultado bastante decente en lo relacionado con la identificación y clasificación de los objetos en cualquier posición en la que se encontraran los objetos, siendo incluso capaz de seguir detectando las cajas con intervención externa de manos para cambiar la posición de las mismas, como se ve en el minuto 3:07. Pese a ello, la baja tasa de *frames* por segundo provoca parones en la imagen y hacen que sea un vídeo algo incómodo de visualizar. A partir del minuto 5:20 la tasa de imágenes por segundos es superpuesta por encima de la imagen de la cámara por medio de una terminal transparente que se va actualizando, la cual indica un valor de alrededor de 5 *fps* la mayor parte del tiempo, con picos máximos de 5.2 y mínimos de 4.9 *fps*.

Un aspecto importante que se puede apreciar en los vídeos es que la predicción no está sujeta únicamente a una imagen, por lo que, si la red se equivoca en su clasificación, en *frames* siguientes donde la imagen cambia ligeramente la red sí es capaz de catalogar el objeto de forma correcta, pudiendo valorar de forma más justa el trabajo desarrollado por la red, comparado con imágenes sueltas.

Los otros 5 vídeos de *tests* que se enlazan junto al descrito han sido grabados primero por la cámara del robot, en dimensiones de 640×480 a una tasa de 30 *fps*, para posteriormente ser evaluados por YOLO y que el resultado obtenido sea fluido. Son vídeos bastante más cortos que el anterior, pero muestran la velocidad a la que podría procesar las imágenes la red si esta fuera ejecutada en un ordenador de altas prestaciones. Mencionar que la calidad visual de estos vídeos es más reducida (480p) debido a que, al ser archivos de duración más corta, la plataforma YouTube los subía automáticamente en esa resolución y no permitía ponerlos en HD.

Por último, la carpeta `darknet` usada para la creación y desarrollo de este proyecto, que contiene los pesos, archivos y ficheros de texto necesarios para su ejecución, tal como se ha explicado y tratado en este documento, han sido subidos al repositorio de **GitHub** personal (*https://github.com/AMendB/tfg-yolo-project*).

5.2 Resumen del proyecto y sus resultados

Los aspectos más destacables de los resultados obtenidos durante el proyecto se podrían sintetizar en los siguientes puntos:

- Todos los objetivos propuestos en el Capítulo 1 sobre la realización del trabajo han sido alcanzados de manera correcta, algunos con mayores dificultades, pero mediante investigación intensa han podido ser solucionados en un plazo lógico.
- El profundo estudio en la temática de Inteligencia Artificial, *Deep Learning* y detectores de objetos en general ha permitido ampliar enormemente los conocimientos para poder dirigir la obra hacia el camino correcto. Gracias a ello se ha podido comprender todos los procedimientos realizados, buscar soluciones en caso de problemas, y obtener unos resultados acordes a lo esperado, pero siempre manteniendo la perspectiva de perfeccionamiento y mejora.
- El rendimiento obtenido durante la ejecución del modelo a tiempo real ofrece una tasa de *frames* estable alrededor de los 5 *fps*, siendo la caja azul la más destacable con una firme precisión media del 98.81 %, mientras que el dato más deficiente lo da el reconocimiento de la caja gris, con un escaso 52.65 % de precisión media. La precisión total de la red teniendo en cuenta todas las clases se sitúa en torno al 73 %, y su *mean average Precision* (mAP@50) en un porcentaje del 77.24 %.
- Los vídeos y pruebas realizados muestran el comportamiento de la red de forma muy visual y ayudan a comprobar hasta qué punto sería aplicable en un vehículo real esta configuración, tema que se tratará en el último capítulo.

6 Conclusiones y Futuros Trabajos

"Abandona los grandes caminos, sigue los senderos."

—Pitágoras

En este último capítulo se llega a la finalización del proyecto realizado como trabajo de fin de grado, aportando una serie de visiones generales, mejoras o evoluciones del mismo, y mostrando varios pensamientos finales que aporten las últimas palabras de lo que ha significado la realización del proyecto.

6.1 Conclusiones

Los resultados obtenidos son considerados acertados para las limitaciones en las que ha sido desarrollada la tarea, con un *dataset* improvisado, un mini ordenador con bajas prestaciones y una cámara de calidad bastante mejorable, se han conseguido un desempeño bastante eficiente y con una precisión adecuada a la situación.

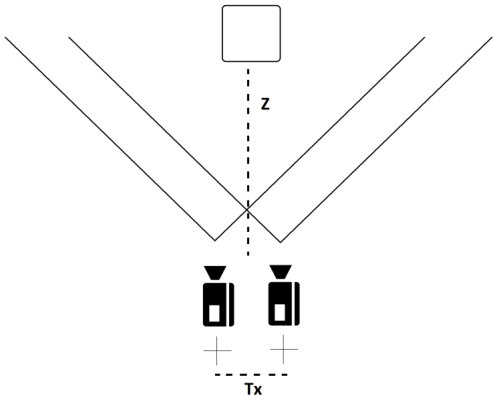


Figura 6.1 Estructura de la configuración del sistema de visión estéreo[1].

[1] Fuente: *https://www.researchgate.net/figure/Figura-1-Configuracion-del-sistema-de-vision-en-estereo_fig1_311643542* [accedido 29 Jun, 2021]

Un aspecto en el que se podría mejorar bastante el funcionamiento es a la hora de detectar cajas del mismo color pero diferente tamaño, como eran las clases *RojoBig* y *RojoSmall*. Al no tener ningún tipo de sensor de proximidad y basarse el modelo únicament en visión artificial, resulta complicado, sobre todo, diferenciar situaciones como cajas pequeñas cercanas de cajas grandes algo más alejadas, ya que al no saber a qué distancia se encuentran de la cámara no sabe su tamaño real. Esto podría reforzarse con el uso de sensores de proximidad o mediante la triangulación y geometría epipolar aplicando visión estéreo con dos cámaras.

Algo que ya se estudió en el Capítulo 5 es la dificultad que ha tenido la red para detectar los objetos más pequeños, sobre todo la caja gris, que es la de tamaño más reducido, pero esta deficiencia es atribuida especialmente a la propia red YOLO, ya que es una problemática que se ha estado mencionando desde su primera versión. Otra circunstancia que ha generado especial confusión en la red es la caja roja rayada, la cual dada las bajas prestaciones de la cámara no se podían apreciar las líneas en muchas de las imágenes. Esta dificultad podría haberse disminuido aumentando el grosor de las líneas para poderse distinguir mejor, pero haría que la dificultad y complejidad en la que se ha querido poner al modelo del proyecto disminuyera. Además, con ello se ha demostrado hasta qué punto puede usarse o no un sistema con una configuración similar en aplicaciones reales, donde las condiciones en las que se encuentran los objetos a identificar son muy dispares e imprevisibles, y las condiciones de visión pueden ser mucho más complejas, con mala iluminación, sombras, diferentes colores, etc.

Se piensa, por tanto, que los resultados podrían haber sido superiores con equipamiento de características más elevadas, como podría ser una cámara con un mejor sensor, o varias de ellas para visión estéreo, un mini ordenador más potente, como puede ser la Nvidia Jetson Xavier NX, la cual permitiría correr la red neuronal a una tasa mayor y más estable de *frames* y estar más cercano a lo que podría ser una implementación para usar en la vida real, o también podría haberse reducido en gran medida el tiempo de entrenamiento con un ordenador con GPU más potente a la que se disponía. Por estas razones se considera que la configuración usada es aceptable para este proyecto concreto, pero quedaría muy alejada para una implementación real en un vehículo autónomo.

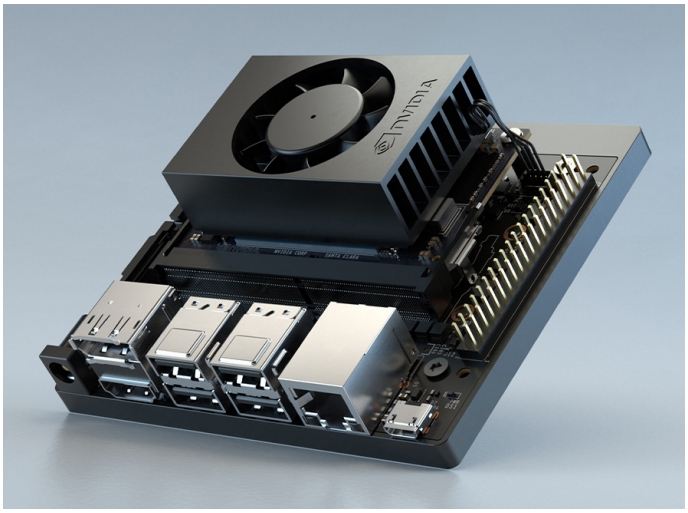


Figura 6.2 Imagen renderizada de la Nvidia Jetson Xavier NX[2].

[2] Fuente: *https://www.nvidia.com/content/dam/en-zz/Solutions/intelligent-machines/jetson-xavier-nx/autonomous-machines-embedded-systems-jetson-xavier-nx-hero-2560-ud@2x.jpg* [accedido 29 Jun, 2021]

6.2 Futuros trabajos

Con vista a futuros trabajos, el primer cambio que se introduciría sería el uso de obstáculos reales en lugar de cajas que los simulen. Esto daría un resultado mucho más ajustado a su uso en un vehículo autónomo real, ya sea acuático, como se planteó en primer lugar, o terrestre, como se ha desarrollado este proyecto.

Si el caso de trabajo es un barco autónomo, este debe disponer de numerosas cámaras gestionadas con IA para la detección de los obstáculos que pueden ser barcos, olas demasiado grandes, piedras, salientes, rompeolas, tierra, escombros o boyas entre otros. Por el contrario, si se trata de un coche autónomo deben tenerse en cuenta muchos más factores, ya que circula por vías públicas en las que hay normas más establecidas y personas muy próximas, por lo que sería necesaria la identificación de señales de tráfico, semáforo, reglas de circulación, así como la detección de obstáculos que pueden ser peatones, bicicletas, otros coches, árboles caídos o calles en obras, entre otras cosas. Todo esto debe realizarse de forma extremadamente rápida y precisa, ya que el tiempo es crucial a altas velocidades, y los errores no pueden permitirse.

Un punto adicional, sin que sea de una dificultad tan elevada como los proyectos mencionados, sería la combinación de la visión artificial desarrollada para este proyecto con el control autónomo del propio JetBot en el que ha sido montada la plataforma Jetson Nano y la cámara. Este dispone de emplazamientos para la colocación de baterías recargables que alimenten tanto a la Jetson como al puente en H que gestione el movimiento de los motores de las dos ruedas motrices de las que dispone, ya que las otras dos de las que dispone son ruedas locas metálicas. Con esta combinación la red sería capaz de detectar los obstáculos existentes en el entorno y tener la posibilidad de trazar rutas que los esquiven, pudiendo aplicar algoritmos de búsqueda como *A** o *Dijkstra*. Una idea adicional realmente interesante podría ser la construcción de laberintos en los que introducir el robot móvil, y mediante la fusión de sensores, visión artificial y control del vehículo pueda conseguir salir del mismo de forma autónoma.

Para finalizar, cabe comentar que la realización de este proyecto fue enfrentada como un reto tanto personal como académico, ya que la inteligencia artificial era un tema que no había sido tratado durante la carrera, y llevarlo a cabo ha supuesto un gran aprendizaje sobre una cuestión cada vez más trascendental y que supone una de las mayores evoluciones recientes.

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] *Aprendizaje por Refuerzo | Aprende Machine Learning*, https://www.aprendemachinelearning.com/aprendizaje-por-refuerzo/, [accedido 4 Jun, 2021].

[2] *Autonomous Vehicles: The Mix of Opportunity and Uncertainty*, https://www.fierceelectronics.com/components/autonomous-vehicles-mix-opportunity-and-uncertainty, [accedido 16 May, 2021].

[3] *Con el nivel 3 de conducción autónoma conduces, pero menos... - Motor.es*, https://www.motor.es/noticias/con-el-nivel-3-de-conduccion-autonoma-conduces-pero-menos-202072774.html, [accedido 14 May, 2021].

[4] *Driving autonomous vehicles forward with intelligent infrastructure*, https://www.smartcitiesworld.net/opinions/opinions/driving-autonomous-vehicles-forward-with-intelligent-infrastructure, [accedido 16 May, 2021].

[5] *El borrador de la ley de conducción autónoma alemana hace más responsable al conductor - Motor.es*, https://www.motor.es/noticias/ley-conduccion-autonoma-alemania-conductor-202175107.html, [accedido 14 May, 2021].

[6] *Inteligencia artificial fácil - Machine Learning y Deep Learning prácticos - El sesgo, una neurona particular | Editiones ENI*, https://www.ediciones-eni.com/open/mediabook.aspx?idR=cb2c5aadd6799b3bffe4bc6930c2faee, [accedido 31 May, 2021].

[7] *Inteligencia Artificial: sus riesgos y cómo proteger tu privacidad*, https://www.lavanguardia.com/tecnologia/actualidad/20181017/452399127153/inteligencia-artificial-riesgos-como-proteger-privacidad.html, [accedido 2 May, 2021].

[8] *La neurona artificial*, https://www.ibiblio.org/pub/linux/docs/LuCaS/Presentaciones/200304curso-glisa/redes_neuronales/curso-glisa-redes_neuronales-html/x38.html , [accedido 28 May, 2021].

[9] *Los niveles de la conducción autónoma*, https://www.cea-online.es/blog/213-los-niveles-de-la-conduccion-autonoma, [accedido 8 May, 2021].

[10] *Modelos de Detección de Objetos | Aprende Machine Learning*, https://www.aprendemachinelearning.com/modelos-de-deteccion-de-objetos/, [accedido 28 May, 2021].

[11] *Opinión| ¿Son los datos el nuevo petróleo del siglo XXI? | Diario TI*, https://diarioti.com/opinionson-los-datos-el-nuevo-petroleo-del-siglo-xxi/109847, [accedido 2 May, 2021].

[12] *Redes neuronales desde cero (I) - Introducción - IArtificial.net*, https://www.iartificial.net/redes-neuronales-desde-cero-i-introduccion/, [accedido 31 May, 2021].

[13] *Target Prediction using Single-layer Perceptron and Multilayer Perceptron | by Ananda Hange | Nerd For Tech | Medium*, https://medium.com/nerd-for-tech/flux-prediction-using-single-layer-perceptron-and-multilayer-perceptron-cf82c1341c33, [accedido 31 May, 2021].

[14] *Un grupo de hackers piratea las más de 200 cámaras que Tesla tiene en China*, https://www.autobild.es/noticias/grupo-hackers-accede-200-camaras-tesla-tiene-china-826731, [accedido 15 May, 2021].

[15] *Usos de la Inteligencia Artificial - Omega2001 Servicios Informáticos*, https://omega2001.es/usos-de-la-inteligencia-artificial/, [accedido 8 May, 2021].

[16] *Understanding AlexNet | Learn OpenCV*, June 2018, https://learnopencv.com/understanding-alexnet/, [accedido 5 Jun, 2021].

[17] *¿qué es machine learning y cómo funciona?*, Mar 2019, https://www.apd.es/que-es-machine-learning/, [accedido 8 Jul, 2021].

[18] *2020: Un año decisivo para los barcos autónomos*, Nov 2020, http://puertosylogistica.com/2020-ano-decisivo-los-barcos-autonomos/, [accedido 1 Jul, 2021].

[19] *Detección de caras*, November 2020, https://es.wikipedia.org/w/index.php?title=Detecci%C3%B3n_de_caras&oldid=131317313, [accedido 4 Jun, 2021].

[20] *Mayflower, el barco autónomo que inicia una misión histórica*, Sep 2020, https://www.elespanol.com/omicrono/tecnologia/20200916/mayflower-barco-autonomo-inicia-mision-historica/521198466_0.html, [accedido 30 Jun, 2021].

[21] *Pioneering norwegian autonomous-ship project receives eu funding*, Jan 2020, https://energynorthern.com/2020/01/22/pioneering-norwegian-autonomous-ship-project-receives-eu-funding/, [accedido 30 Jun, 2021].

[22] *Redes neuronales residuales - Lo que necesitas saber (ResNet)*, May 2020, https://datascience.eu/es/aprendizaje-automatico/una-vision-general-de-resnet-y-sus-variantes/, [accedido 3 Jun, 2021].

[23] *Yara birkeland: Primer buque totalmente autónomo puesto en espera indefinida*, May 2020, https://www.mascontainer.com/yara-birkeland-primer-buque-totalmente-autonomo-puesto-en-espera-indefinida/, [accedido 30 Jun, 2021].

[24] *¿cuál es la diferencia entre el machine learning y el deep learning?*, 2020, https://blog.bismart.com/es/diferencia-machine-learning-deep-learning, [accedido 8 Jul, 2021].

[25] *AlexNet*, May 2021, https://en.wikipedia.org/w/index.php?title=AlexNet&oldid=1021610217, [accedido 4 Jun, 2021].

[26] *Histogram of oriented gradients*, February 2021, https://en.wikipedia.org/w/index.php?title=Histogram_of_oriented_gradients&oldid=1004270806, [accedido 4 Jun, 2021].

[27] *Residual neural network*, April 2021, https://en.wikipedia.org/w/index.php?title=Residual_neural_network&oldid=1019286662, [accedido 3 Jun, 2021].

[28] *Viola–Jones object detection framework*, May 2021, https://en.wikipedia.org/w/index.php?title=Viola%E2%80%93Jones_object_detection_framework&oldid=1021428546, [accedido 4 Jun, 2021].

[29] *Yara birkeland, el primer buque de contenedores autónomo del mundo*, Jun 2021, https://somoselectricos.com/yara-birkeland-primer-buque-contenedores-autonomo/, [accedido 30

Jun, 2021].

[30] José Manuel Anguas Pérez, *Análisis y diseño de dispositivos de radiocomunicación mediante redes neuronales* (spa), *http://bibing.us.es/proyectos/abreproy/11084/fichero/Memoria+por+cap%C3%ADtulos+%252FCap%C3%ADtulo+4.pdf+*, [accedido 29 May, 2021].

[31] Ignacio Arriola Oregui, *Detección de objetos basada en Deep Learning y aplicada a vehículos autónomos*, (2018) (spa), *https://addi.ehu.es/handle/10810/28983*, [accedido 29 May, 2021].

[32] Sergio Canu, *Yolo v3 - install and run yolo on nvidia jetson nano (with gpu)*, Mar 2021, *https://pysource.com/2019/08/29/yolo-v3-install-and-run-yolo-on-nvidia-jetson-nano-with-gpu/*.

[33] Xianzhi Du, Tsung-Yi Lin, Pengchong Jin, Golnaz Ghiasi, Mingxing Tan, Yin Cui, Quoc V. Le, and Xiaodan Song, *Spinenet: Learning scale-permuted backbone for recognition and localization*, CoRR **abs/1912.05027** (2019), *http://arxiv.org/abs/1912.05027*, [accedido 12 Jun, 2021].

[34] EFE, *Google admite escuchar el 0,2 % de las conversaciones con su asistente virtual*, El País (2019) (es), *https://elpais.com/elpais/2019/07/12/ciencia/1562914719_220640.html*, [accedido 2 May, 2021].

[35] ESCOM, *Características de las Redes Neuronales*, *https://es.slideshare.net/mentelibre/caractersticas-de-las-redes-neuronales*, [accedido 2 Jun, 2021].

[36] Sergio C. Fanjul, *¿Nos espían los chismes tecnológicos?*, El País (2020) (es), *https://elpais.com/retina/2020/02/24/innovacion/1582569720_734965.html*, [accedido 2 May, 2021].

[37] Yago Gantes, *Sin tripulación y con inteligencia artificial: así es el barco autónomo de ibm que cruzará por primera vez el atlántico*, Mar 2020, *https://www.eleconomista.es/status/noticias/10409403/03/20/Sin-tripulacion-y-con-Inteligencia-Artificial-asi-es-el-barco-autonomo-de-IBM-que-cruzara-por-primera-vez-el-Atlantico-.html*, [accedido 30 Jun, 2021].

[38] Campanini García and Diego Alejandro, *Detección de objetos usando redes neuronales convolucionales junto con Random Forest y Support Vector Machines*, (2018) (es), *http://repositorio.uchile.cl/handle/2250/167863*, [accedido 30 May, 2021].

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition*, arXiv:1406.4729 [cs] **8691** (2014), 346–361, *http://arxiv.org/abs/1406.4729*, [accedido 12 Jun, 2021].

[40] Amro Kamal, *YOLO, YOLOv2 and YOLOv3: All You want to know | by Amro Kamal | Medium*, *https://amrokamal-47691.medium.com/yolo-yolov2-and-yolov3-all-you-want-to-know-7e3e92dc4899*, [accedido 9 Jun, 2021].

[41] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie, *Feature Pyramid Networks for Object Detection*, arXiv:1612.03144 [cs] (2017), *http://arxiv.org/abs/1612.03144*, [accedido 12 Jun, 2021].

[42] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár, *Focal Loss for Dense Object Detection*, arXiv:1708.02002 [cs] (2018) (en), *http://arxiv.org/abs/1708.02002*, [accedido 11 Jun, 2021].

[43] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg, *SSD: Single Shot MultiBox Detector*, arXiv:1512.02325 [cs] **9905** (2016), 21–37 (en), *http://arxiv.org/abs/1512.02325*, [accedido 9 Jun, 2021].

[44] Javier López and Ecija Abogados, *¿Son compatibles inteligencia artificial y privacidad?*, *https://revistabyte.es/actualidad-it/inteligencia-artificial-privacidad/*, [accedido 2 May, 2021].

[45] Jose Martinez Heras, *¿clasificación o regresión?*, Sep 2020, *https://www.iartificial.net/clasificacion-o-regresion/#Clasificacion*, [accedido 8 Jul, 2021].

[46] Juan Miguel Marín Diazaraque, *Introducción a las Redes Neuronales Aplicadas*, 2019, *http://halweb.uc3m.es/esp/Personal/personas/jmmarin/esp/DM/tema3dm.pdf*, [accedido 2 Jun, 2021].

[47] Claudio Moreno, *Buques autónomos: "se venden como una mejora pero supondrá la debacle del sector"*, *https://www.equaltimes.org/buques-autonomos-se-venden-como?lang=es*, [accedido 30 Jun, 2021].

[48] Ramón Muñoz, *Multa a LaLiga porque su aplicación usaba el micrófono del móvil para cazar bares 'piratas'*, El País (2019) (es), *https://elpais.com/economia/2019/06/11/actualidad/1560264403_529943.html*, [accedido 2 May, 2021].

[49] Eduardo Peris, *¿Funcionará el coche autónomo en el mundo rural?*, April 2019, *https://hackercar.com/funcionara-el-coche-autonomo-en-el-mundo-rural/*, [accedido 15 May, 2021].

[50] Jordi Gisbert Ponsoda, *No tienes escapatoria, Google te entiende al 95 %*, *https://andro4all.com/2017/06/google-entiende-95-lenguaje-humano*, [accedido 2 May, 2021].

[51] Alberto Pérez, *El mayflower autonomous ship eléctrico tiene problemas y regresa a inglaterra*, Jun 2021, *https://www.hibridosyelectricos.com/articulo/navegacion-sostenible/mayflower-autonomous-ship-electrico-tiene-problemas-regresa-inglaterra/20210629182146046503.html*, [accedido 30 Jun, 2021].

[52] _____, *El mayflower autonomous ship será el primer barco eléctrico autónomo en cruzar el atlántico*, Jun 2021, *https://www.hibridosyelectricos.com/articulo/actualidad/mayflower-autonomous-ship-sera-primer-barco-electrico-autonomo-cruzar-atlantico/20210604101723045690.html*, [accedido 30 Jun, 2021].

[53] Joseph Redmon, *Yolo: Real-time object detection*, 2018, *https://pjreddie.com/darknet/yolo/*, [accedido 8 Jul, 2021].

[54] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, arXiv:1506.02640 [cs] (2016) (en), *http://arxiv.org/abs/1506.02640*, [accedido 7 Jun, 2021].

[55] Elaine Rich and Kevin Knight, *Inteligencia artificial*, McGraw-Hill, Madrid, 1994.

[56] Mariano Rivera, *resnet.md*, *http://personal.cimat.mx:8181/~mrivera/cursos/aprendizaje_profundo/resnet/resnet.html*, [accedido 2 Jun, 2021].

[57] Adrian Rosebrock, *Intersection over Union (IoU) for object detection*, November 2016, *https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/*, [accedido 8 Jun, 2021].

[58] Adrian Rosebrock, *Getting started with the NVIDIA Jetson Nano*, May 2019, *https://www.pyimagesearch.com/2019/05/06/getting-started-with-the-nvidia-jetson-nano/*, [accedido 17 May, 2021].

[59] Adrian Rosebrock, *How to configure your NVIDIA Jetson Nano for Computer Vision and Deep Learning*, March 2020, *https://www.pyimagesearch.com/2020/03/25/how-to-configure-your-nvidia-jetson-nano-for-computer-vision-and-deep-learning/*, [accedido 17 May, 2021].

[60] Carlos Alberto Ruiz, Marta Susana Basualdo, and Damián Jorge Matich, *Redes Neuronales: Conceptos Básicos y Aplicaciones.*, 55 (es), *https://www.frro.utn.edu.ar/repositorio/catedras/quimica/5_anio/orientadora1/monograias/matich-redesneuronales.pdf*, [accedido 2 Jun, 2021].

[61] Petru Soviany and Radu Tudor Ionescu, *Optimizing the Trade-off between Single-Stage and Two-Stage Object Detectors using Image Difficulty Prediction*, arXiv:1803.08707 [cs] (2018) (en), *http://arxiv.org/abs/1803.08707*, [accedido 7 Jun, 2021].

[62] Chamidu Supeshala, *YOLO v4 or YOLO v5 or PP-YOLO? Which should I use? | Towards Data Science*, *https://towardsdatascience.com/yolo-v4-or-yolo-v5-or-pp-yolo-dad8e40f7109*, [accedido 9 Jun, 2021].

[63] Jordi Torres, *Deep learning : introducción práctica con keras : primera parte*, Kindle Direct, Barcelona, 2018.

[64] Aurélien Vannieuwenhuyze, *Inteligencia artificial fácil : machine learning y deep learning prácticos*, ENI, 2020.

[65] Jerry Wei, *AlexNet: The Architecture that Challenged CNNs*, September 2020, *https://towardsdatascience.com/alexnet-the-architecture-that-challenged-cnns-e406d5297951*, [accedido 4 Jun, 2021].

[66] Shivy Yohanandan, *map (mean average precision) might confuse you!*, Jun 2020, *https://towardsdatascience.com/map-mean-average-precision-might-confuse-you-5956f1bfa9e2*, [accedido 29 Jun, 2021].

[67] Diego Yriarte, *Mayflower, un barco solar autónomo en busca de conocimiento*, Sep 2020, *https://www.hibridosyelectricos.com/articulo/navegacion-sostenible/mayflower-barco-solar-autonomo-busca-conocimiento/20200926164321038455.html*, [accedido 30 Jun, 2021].

[68] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye, *Object Detection in 20 Years: A Survey*, arXiv:1905.05055 [cs] (2019) (en), *http://arxiv.org/abs/1905.05055*, [accedido 28 May, 2021].